The Zeus Agent Building Toolkit

ZEUS Methodology Documentation Part III

The Application Realisation Guide

Jaron Collis, jaron@info.bt.co.uk
Divine Ndumu, ndumudt@info.bt.co.uk
Intelligent Systems Research Group, BT Labs

Release 1.0, May 1999

Index

1 INTRODUCTION 4

1.1 THE CONCEPTUAL ZEUS AGENT 4
1.2 AN OVERVIEW OF THE AGENT REALISATION PROCESS 5
1.3 GETTING STARTED 6

2 THE ONTOLOGY CREATION STAGE 8

ONT-1: How to Create a New Ontology 8
ONT-2: How to Load an Ontology 8
ONT-3: How to Create a New Fact 9
ONT-4: How to Create a New Attribute 10
ONT-5: How to Set the Type of an Attribute 10
ONT-6: How to Restrict an Attribute 11
ONT-7: How to Set Attribute Defaults 11
ONT-8: How to Create a Restriction 11
ONT-9: How to Save an Ontology 12
How to Merge Ontologies 12

3 THE AGENT CREATION STAGE 13

Choosing the Application Time-Grain 13
Getting Started 14
How to Create an Agent 14
How to Clone an Agent 14
How to Delete an Agent 14
How to Save an Agent 15
How to Configure an Agent 15
3.1 THE AGENT DEFINITION STAGE 16
DEF-1: Configuring Planning Parameters 16
DEF-2: Task Identification 17
DEF-3: Initial Agent Resource Allocation 18
3.2 THE TASK DEFINITION STAGE 19
TASK-1: Using the Primitive Task Editor 19
TASK-2: Using the Summary Task Editor 23
3.3 THE AGENT ORGANISATION STAGE 24
ORG-1: Entering Known Acquaintances 24
ORG-2: Entering Known Abilities 25
3.4 THE AGENT CO-ORDINATION STAGE 27
An Insight into Agent Interactions 27
COORD-1: How to Equip an Agent with a Co-ordination Protocol 29
COORD-2: How to Equip an Agent with an Interaction Strategy 30
COORD-3: How to add new Interaction Protocols and Strategies 32

4 THE UTILITY AGENT CONFIGURATION STAGE 33

UTIL-1: Configuring the Name Servers 33
UTIL-2: Configuring the Facilitators 35
UTIL-3: Configuring the Visualisers 36
UTIL-4: Configuring the Database Proxies 37

5 THE TASK AGENT CONFIGURATION STAGE 38

TACF-1: Configuring the Task Agents 38

6 THE AGENT IMPLEMENTATION STAGE 40

IMPL-1: How to Generate Task and Agent Source Code 40
IMPL-2: How to Implement a Task 42

IMPL-3: How to Connect an Agent to an External Resource *44*
IMPL-4: How to Connect an Agent to an External Program *46*
IMPL-4A: How to Instruct an Agent to Do Something *46*
IMPL-4B: How to Influence an Agent's Behaviour *48*
IMPL-4C: How to Access an Agent's Resources *49*
IMPL-5: How to Implement a new Interaction Strategy *50*
RUNNING THE APPLICATION 51
CONCLUDING REMARKS 51

1 INTRODUCTION

The agent realisation process follows the design stage of the ZEUS agent development methodology. During the course of this document the realisation process will be methodically described is as a series of development stages, each of which consists of individual activities that implement particular aspects of an agent.

A prerequisite for using this document is a design for the application being realised. In addition the developer should know when and how to perform the necessary activities. We have tried to capture this expertise in the Case Studies document, which provides worked examples that we feel best illustrate how to implement working agent applications.

But before we consider how to implement an agent, we need to consider the conceptual basis of agents created with the ZEUS toolkit.

1.1 The Conceptual Zeus Agent

At the highest level of abstraction, the ZEUS agent design approach requires developers to view an agent as a five-layered entity, as illustrated in Figure 1.1 below.

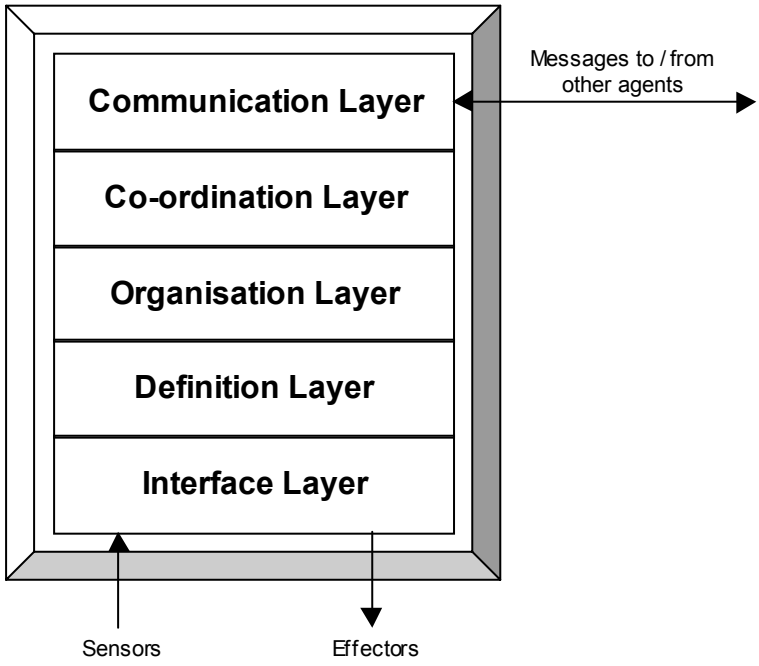


Figure 1.1: The Conceptual Structure of a ZEUS Agent

From the bottom up, these layers are:

- an interface layer, which enables the agent to be linked to the external programs that provide it with resources and/or implement its competencies
- a *definition layer*, where the agent is viewed as an autonomous reasoning entity
- an *organisation layer*, where the agent is viewed in terms of its relationships with other agents
- a *co-ordination* layer, where the agent is viewed as a social entity that interacts according to its known protocols and strategies
- a communication layer, which implements the protocols and mechanisms that support inter-agent communication

The *Interface* layer receives input via its sensors and is able to change the outside world through its effectors. For instance, a sensor could receive instructions from a GUI or act by modifying a database, thus providing physical realisations of the agents' resources and skills. As the sensors and effectors are external to the agent, configuration of the Interface layer consists of specifying interfaces between them and the agent.

At the *Definition* layer the agent is viewed as an autonomous rational entity, in terms of its reasoning abilities, goals, resources, skills, beliefs and preferences. As this layer is physically realised by classes from the ZEUS Agent Component library, (described in the Technical Manual), this stage of the methodology involves the configuration of these components.

At the *Organisation* layer the agent is viewed in terms of its relationships with other agents. This introduces the concept of an agency - a group of related agents. Agencies may be real, in that they are related by virtue of common attribute, such as being part of the same company. Or agencies may be virtual, in that their constituents share a co-operation agreement. Thus this stage of the methodology involves configuring the agent in terms of the agency it belongs to, what roles it plays with the agency, what other agents it is aware of, and what abilities it knows others possess.

At the *Co-ordination* layer the agent is considered to be a social entity, hence this stage involves configuring it for the desired forms of social behaviour. This is achieved by equipping it with the appropriate negotiation protocols and strategies.

The *Communication* layer provides a transport protocol and language enabling agents to send messages to each other. For compatibility this layer should be the same in each agent, and so there no need for developers to configure this layer.

This document will explain the process by which this conceptual agent is configured into functional member of a multi-agent society. This configuration process will be accomplished using the ZEUS Agent Generator tool, which will ultimately generate the agent source code. The developer need only implement the agents' application-specific functionality, and link it to the agents using the interfaces provided.

1.2 An Overview of the Agent Realisation Process

The objective of this process is to realise working agent implementations from conceptual designs created during the previous stage. The agent realisation process consists of several stages, (some of which have been derived from the levels of abstraction that exist within a ZEUS agent as shown in Figure 1.1). The stages and the order in which they should be attempted are:

Stage 1: Ontology Creation

Before implementing any agents the developer must define the application ontology: the declarative knowledge that represents the significant concepts within the application domain. The tool used to enter this information is the ZEUS Ontology Editor. Or alternatively, an existing ontology can be imported.

Stage 2: Agent Creation

During this stage the generic ZEUS agent is configured to fulfil its application-specific responsibilities, returning in a task agent. This process involves the ZEUS Agent Editor to complete up to four sub-stages (depending on the nature of the agent); these are:

- Agent Definition - where its tasks, initial resources and planning abilities are specified
- Task Description - where the applicability and attributes of agent activities are specified
- Agent Organisation - where the social context of each agent is specified
- Agent Co-ordination - where each agent is equipped with the social abilities for interaction

Stage 3: Utility Agent Configuration

Whereas the previous section concerned the agents who performed the application-specific activities this stage defines the attributes of the utility agents who provide the support infrastructure for the agent society. This information is entered through the Code Generation Editor in readiness for the creation of the utility agents.

Stage 4: Task Agent Configuration

This stage enables the runtime parameters of the task agents to be specified. This involves supplying information such as the host machines the agents will run on, and the external resources and programs to which the agents will be linked.

Stage 5: Agent Implementation

At this stage the Code Generator can be invoked and agent source code automatically generated. This leaves the developer with the job of providing the application-specific implementations of the tasks, external resources, programs (such as agent user interfaces) and interaction strategies. When this stage has been completed the application is ready to be run.

Each of these stages is the subject of the remaining sections of this document. Each section contains various activities that describe how to configure aspects of the generic ZEUS agent. For instance, the Co-ordination section describes how to solve the problem of engaging in a dialogue with another agent (the solution being to equip agent with an appropriate co-ordination protocol).

These problem/solution entries are referred to in the Case Studies that accompany the Role Modelling Guide. If you are reading this document for the first time it is recommended that you read this guide and one of the associated case studies before proceeding further.

1.3 Getting Started

All the stages of the realisation process are achieved through the ZEUS Agent Generator tool. The Generator is run through the Java runtime environment, which should be version 1.2 or later (due to the class libraries required). The presence of a JRE can be easily tested by typing 'java -version' into your machine's command line, if it is installed correctly you will see a message reporting the version of the JRE installed on that machine. Otherwise you will need to install the JRE (or make it accessible) before proceeding.

You should also ensure that the JRE's CLASSPATH environment variable contains a reference to the directory that contains the zeus class package. The state of this variable can be seen by typing SETENV on UNIX machines, or SET on PCs. There should also be a ZEUS properties file in the appropriate location, (for more detailed instructions, see the Installation guide).

- Now start the Generator by entering the following command:

  ⇒ java zeus.generator.AgentGenerator

If ZEUS has been installed properly a window should appear entitled 'ZEUS Agent Generator'. If not, consult the trouble-shooting section of the installation guide.

The Generator window serves as the launching point for the various agent building windows, it consists of four panels that enable access to its functions, namely:

⇒ **Project Options** - this panel contains a toolbar of management options, enabling whole projects to be loaded, saved and cleared. The other buttons launch the Society Viewer and Code Generator tools. Beneath is the filename of the project currently in memory.

⇒ **Ontology Options -** this panel provides access to the Ontology Editor tool, enabling ontologies to be loaded, saved, cleared and edited. The name of the ontology currently in memory is shown beneath the toolbar.

⇒ **Agent Options** - this panel consists of a toolbar and a table that lists all the agents in the currently loaded project. The options available from here are described in Section 3 of this document.

⇒ **Task Options** - this panel consists of a toolbar and a table that lists all the tasks in the currently loaded project. The options available from here are also described in Section 3.

The remainder of this document will describe how the facilities of the Generator tool support the stages of the ZEUS agent creation methodology.

2 THE ONTOLOGY CREATION STAGE

An ontology is a set of declarative knowledge representing every significant concept within a particular application domain. The significance of a concept is easily assessed, if meaningful interaction can not occur between agents without both parties being aware of it, then the concept *is* significant and must be modelled. Note that for convenience we use the term 'fact' throughout ZEUS to describe an individual domain concept.

Prior to attempting this stage you should have already identified the following:

- the key concepts within the problem domain
- the significant attributes of each concept
- the types of each attribute
- any constraints on the attributes

The remainder of this section describes how to enter this information into the ZEUS Ontology Editor. This can be launched from the ZEUS Agent Generator tool, which displays the following panel:

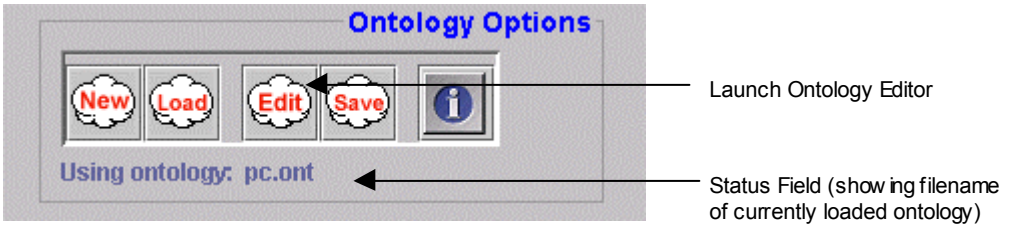


Figure 2.1: A screenshot of the Ontology Options panel of the Generator tool

---

ONT-1: How to Create a New Ontology

- If the Ontology Editor is not running, click on the "Edit" button in the Generator tools' `Ontology Options` toolbar, or select the "New Ontology" menu option from the `Ontology Menu`. This will create a new (empty) ontology and launch the Ontology Editor. From now onward all facts entered through this editor will become part of this ontology.

- If the Ontology Editor is already running, select the "New" menu option; if there is currently an ontology in memory you will be asked whether you want to save it. Clicking on the "New" button in the Generator tools' `Ontology Options` toolbar will clear the current ontology without opening the Editor.

ONT-2: How to Load an Ontology

- If the Ontology Editor is not running, click on the "Load" button in the Generator tools' `Ontology Options` toolbar, or select the "Load Ontology" menu option from the `Ontology Menu`.

- If the Ontology Editor is already open, a new ontology can be loaded by choosing the "Load…" menu option.
- This will summon a file requester, locate the file containing the Ontology and click "OK". If there is already an ontology in memory, you will be asked if you want to save it before it is removed.

When the ontology has been loaded and parsed, its contents will appear in the Ontology Editor's windows. A screenshot of the Ontology Editor is shown in Figure 2.1.

Note that the fact pane of the Ontology Editor is divided into two, an upper panel that displays the Fact Hierarchy, and a lower panel that displays the attributes of the currently selected fact. The panels are separated by a split-pane that can be moved up or down to enlarge one panel at the expense of the other.

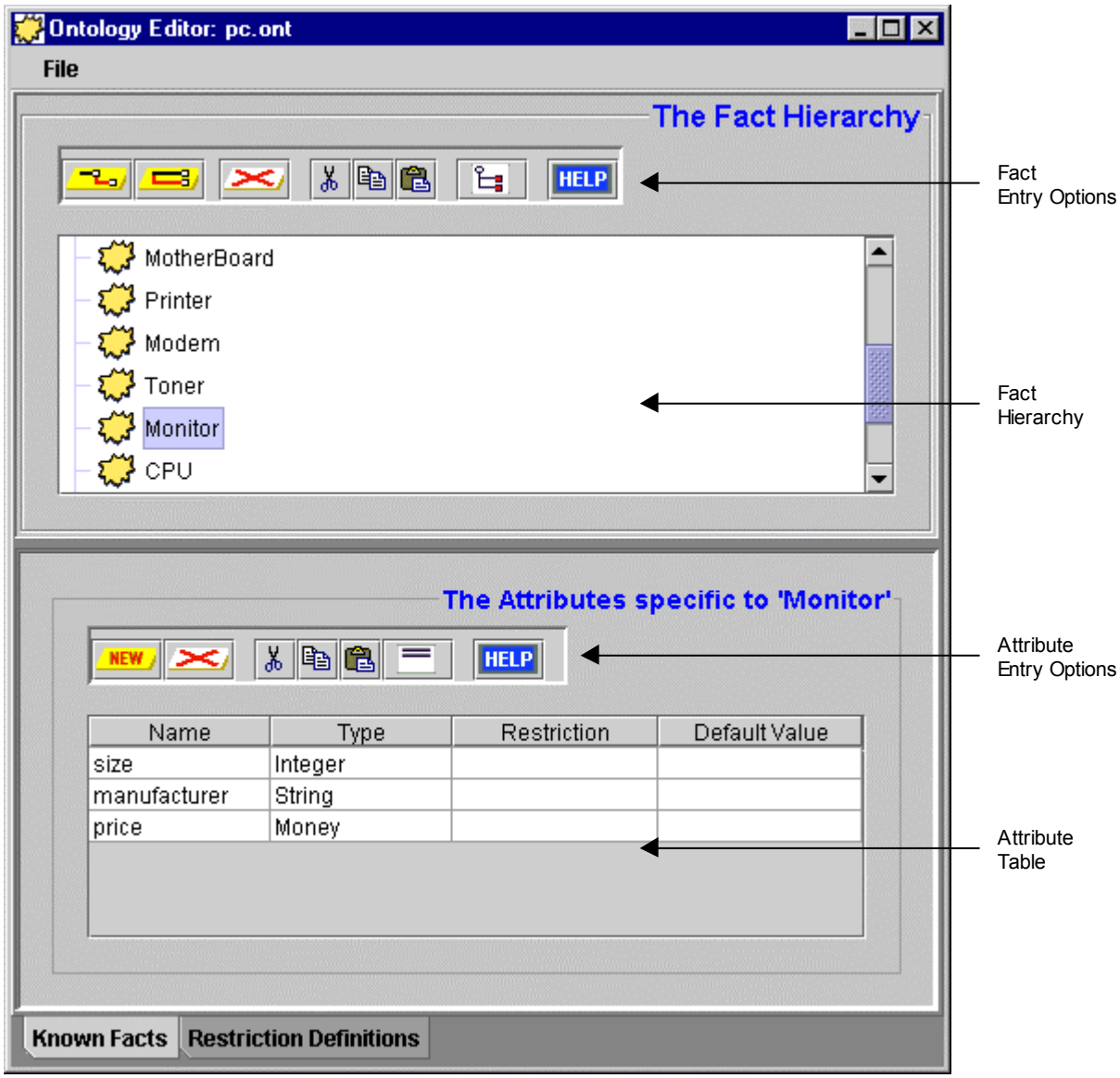


Figure 2.2: A Screenshot of one of the panels of the Ontology Editor. Notice how the one of the nodes in the fact hierarchy (Monitor) has been selected, and its attributes are being shown in the table below.

ONT-3: How to Create a New Fact

The first stage in creating a new fact is to determine its position within the existing fact hierarchy:

- If the new fact has no parents in the existing hierarchy you will typically choose the **Entity** node and click on the "Add New Child Fact" button.

The **Entity** fact is used to represent concepts that have physical realisations, like objects and commodities. Where this is not appropriate the ZEUS fact hierarchy also has a fact called **Abstract**

that does not possess cardinality and value attributes.

- If the new fact inherits from an existing fact, select the parent node and click on the "Add New Child Fact" button.
- To make the new fact share the same parent as another, (a useful short cut), select the existing fact and click the "Add New Peer Fact" button.
- A new entry will now appear in the fact tree, double click on it and rename to whatever is appropriate.

ONT-4: How to Create a New Attribute

- Ensure that the fact to which the attribute will be added is the selected node of the fact hierarchy, if not, clicking on the fact selects it. The existing attributes for the selected fact will then appear in the Attribute Table beneath the Fact Hierarchy window.
- Click on the "New" button in the toolbar above the Attribute Table. This will create a new entry in the table, double-click on it to rename it as appropriate.
- Clicking on the 'Toggle Shown Attributes' will hide or show the attributes inherited from the fact's parents. This can be useful for determining whether an attribute has already been declared higher in the fact hierarchy.

ONT-5: How to Set the Type of an Attribute

Types serve to restrict the set of valid values the attribute can take, and will be one of the basic types, a restriction or another fact.

- Double-click on an entry in the Attribute Table, this will summon a window showing all a hierarchy of currently known types, as shown in Figure 2.3.

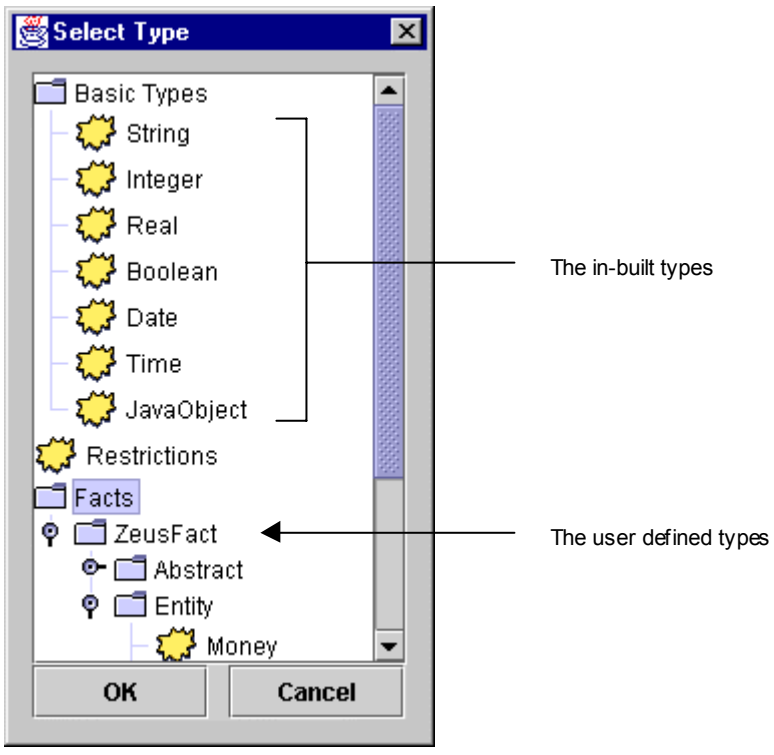


Figure 2.3: A screenshot of the Type Selection window

The basic types have been predefined, they are:

- ⇒ **String**: any combination of alpha-numeric characters, plus the _ and $ characters *only*
- ⇒ **Integer**: whole numbers in the range -2147483648 to +2147483647, (these are the minimum and maximum integer values in Java).
- ⇒ **Real**: all numbers in the range $4.94e^{-324}$ to $1.798e^{308}$, (these are the minimum and maximum double values in Java).
- ⇒ **Boolean**: the words *true* or *false*
- ⇒ **Date**: numerically expressed in the European format: dd/mm/yyyy, (i.e. 2 digits for day/ 1 or 2 for month/4 for year). Terms can also be separated by dashes, i.e. dd-mm-yyyy. ZEUS is millennium compliant.
- ⇒ **Time**: numerically expressed using the 24-hour clock format, as either hh:mm:ss, (i.e. 1 or 2 digits for each of hours:minutes:seconds). Seconds can be omitted, and hh:mm used instead.
- ⇒ **JavaObject**: an opaque reference to a runtime Java object

Restrictions are custom types used to further restrict the values possible for basic types. These are entered separately, see 'How to Create a Restriction'. All previously entered restrictions are listed under the "Restrictions" node of the type selection window.

Where attributes need to assume more complex values, their type can be specified as a particular fact. All previously entered facts are listed under the "Facts" node of the type selection window.

ONT-6: How to Restrict an Attribute

- An alternative to creating a restriction rule is to restrict each attribute individually, this is achieved by double clicking on the "Restriction" cell of the attribute entry, which opens a small green-coloured panel through which the restriction can be typed.
- The syntax of restrictions is that of a simplified expression, full details are in the Appendix.

ONT-7: How to Set Attribute Defaults

- To give an attribute a default value, double click on the "Default" cell of the attribute entry, this opens a small green-coloured panel through which the default value can be typed.
- The syntax of the value is dependent on the attribute type. Defaults can not be set for attributes whose type is a restriction or another fact.

ONT-8: How to Create a Restriction

Restrictions are types that have been constrained to a particular set or range of values. Although it is possible to restrict attributes without creating a restriction (see 'How to Restrict an Attribute'), restrictions allow for reuse, and provide meaningful names. Restrictions are entered through the 'Restriction Definitions' pane of the Ontology Editor.

- To create a restriction click on 'Restriction Definitions' tab at the bottom of the Ontology Editor. This shows the list of known restrictions (initially empty).
- Add a new restriction by clicking on the "New" button on the toolbar, this will create a new entry in the Restriction Table, double click on it to edit its name to something more appropriate.
- Now set the type of the restriction by double clicking on its "Type" cell, this brings up a window with a hierarchy of basic types (see 'How to Set the Type of an Attribute' for details) and current restrictions. Facts can not be chosen as a type.

- Now double click on the "Restriction" cell, this opens a small green-coloured panel through which the default value can be typed. The syntax will be dependent on the type chosen previously.
- The restriction will be syntactically checked upon entry, invalid entries will be displayed in red.

ONT-9: How to Save an Ontology

Ontologies are currently saved in an ASCII file using our own format; saving ontology in a more standardised format, like XML is under consideration.

- If the Ontology Editor is not open, click on the "Save" button in the Generator's `Ontology Options` toolbar, or select the "Save Ontology" menu option from the `Ontology Menu`.

- If the Ontology Editor is still open, choosing the "Save" menu option will save its contents.
- If the ontology is unnamed a file requester will appear, and you will be prompted for a filename to save it under.
- If the ontology has already been named it will be saved under that name. To save it under a different name, choose the "Save Ontology As…" menu option.

How to Merge Ontologies

- This function is not yet supported in the current Ontology Editor.

3 THE AGENT CREATION STAGE

During the agent creation stage the generic ZEUS agent is configured to fulfil its application-specific responsibilities. Thus by the time this stage is attempted the following design decisions should have already been taken:

- What is the granularity of time for the application?
- What agents exist?
- What activities will each agent perform?
- How will each agent interact with other agents?
- What strategies and expertise does each agent know?

Guidance for these agent design decisions is provided in the Role Modelling guide and its case studies. This is intended to help the developer decide which agent aspects need to be reconfigured, according to the type of agent being created.

Choosing the Application Time-Grain

But before agents are defined the developer must decide on the application's granularity of time. The granularity refers to the smallest possible indivisible period of time for the application: a period known the time-grain. The time-grain determines the period between 'ticks' of the application-wide clock that is maintained by the Agent Name Servers. This global clock acts like Greenwich Mean Time (GMT), providing a standard time reference for all agents that is independent of any one agent's own local time. It thus provides a means of synchronising agents regardless of their physical distribution.

In ZEUS the duration of all agent activities is expressed in time-grains rather than seconds or minutes, making the choice of time-grain length a significant decision. The key factor is how rapidly the agents' environment changes. This is because new activities are started at the beginning of a time-grain, meaning there will be a delay between an event occurring and the agent reacting.

So, for instance, if the time-grain is 1 minute in length, and an event is detected 5 seconds after the time-grain begins, the agent will not react for at least another 55 seconds. There is another implication, if an agent can perform one activity per time-grain, and this activity takes on average 1 second to complete, it will be idle for 59 seconds in every minute.

From this discussion one might think that long time-grains are best avoided. However, because the time-grain is used for synchronisation there are dangers in choosing too small a duration. For instance, imagine that the time-grain is set to 1 second, this will only give each agent 1 second to observe its environment, make decisions and potentially perform actions. This may pose no problems for an agent on a fast machine that has no communication latency, but such performance is difficult to guarantee. The consequences of agents slipping out of synchronisation can be quite serious. Agents may begin to unexpectedly miss deadlines and the application may cease working or behave unpredictably, (and the cause may prove very difficult to undercover).

The choice of time-grain is thus dependent on factors like the nature of the application, the reliability of the network and the speed of the host machines that support the agents. For instance, a value of 30 seconds may be too slow for an application like network management, which tends to involve a very dynamic environment, or where agents need to continually interact with human users. Likewise if agents were monitoring changes to a relatively static subject, like a web site, a longer time-grain would probably be more efficient.

The default value for ZEUS applications is 30 seconds, which in our experience is long enough to allow several agents to run on the same machine and still respond comfortably.

For instructions on how to change the time-grain for your application see Section 4: "Utility Agent Configuration".

Getting Started

The agent creation options are accessible through the 'Agent Options' panel of the ZEUS Agent Generator tool, which is illustrated in Figure 3.1.

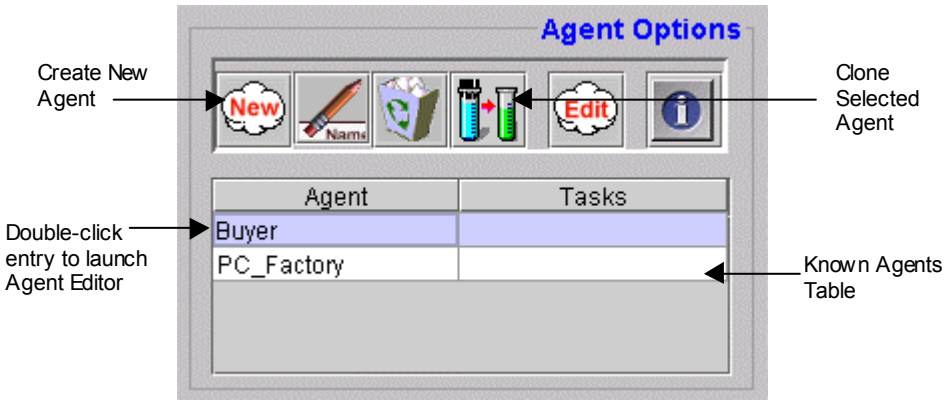


Figure 3.1: A screenshot of the Agent Options Panel of the Generator tool. The Known Agent Table lists all the agents that have been defined in the current project.

How to Create an Agent

The Agent Generator tool does not treat agents as classes in the conventional object oriented sense, where each class name refers to a template that can create as many instances as needed at run-time. Instead, each agent instance refers to an individual entity, and must be given its own unique name when it is created. This is important to enable social relationships to be defined in terms of individuals, rather than classes of agents.

- To create a new agent, select the `New` button from the `Agent Options` toolbar. This will create a new entry in the agent table using a temporary name.

- Give the agent a more informative name by selecting its entry in the table and clicking on the `Rename Agent` button. The agent's name field will be editable, type in the new name and then press <return>.

How to Clone an Agent

If more than one agent shares the same expertise, abilities and attributes the developer will have to create a separate entry for each. Rather than re-entering the same information, once an individual has been defined it can be cloned as a separate individual and given its own unique name.

- This is achieved by selecting the agent to clone in the table and clicking on the `Clone Agent` button from the `Agent Options` toolbar. This creates a new entry in the table using a temporary name, which can be renamed to something more appropriate.

How to Delete an Agent

- An agent entry can be deleted permanently from the project by selecting its entry and clicking on the `Delete` button from the `Agent Options` toolbar. If you are certain you want to delete this agent, click 'OK' when the confirmation requester appears.

How to Save an Agent

In the current version of the Generator there is no option to save agent definitions separately, only as part of a project.

- Saving the project is achieved by clicking on the `Save` button in the `Project Options` toolbar, or by choosing the `Save Project` menu option.

How to Configure an Agent

Once an agent has been created it can be configured to meet the requirements of its intended role.

- To edit an agent either double clicking on its name in the Agent Table, or selecting its entry and click on the `Edit` button in the `Agent Options` toolbar.
- This opens a new window entitled 'Agent Editor' containing 3 separate tab-panes that each facilitate the entry of information related to the methodology stage of the same name.

The remainder of this section describes the different sub-stages of the Agent Creation process, and how the Agent Editor supports them.

3.1 The Agent Definition Stage

This aspect of the methodology is performed through the 'Agent Definition' panel of the Agent Editor. The Agent Definition Stage consists of three main activities, these are:

⇒ Configuring Planning Parameters, see activity **DEF-1**

⇒ Task Identification, see activity **DEF-2**

⇒ Initial Agent Resources, see activity **DEF-3**

A useful metaphor at this stage is to view the agent as a manager in charge of a factory that comprises a number of production lines (of identical machines), each of which can perform a number of different tasks. The manager possesses some resources, and the production of an item may consume some of these resources, with the production process lasting for a finite time interval.

The role of the manager is to produce items at the request of customers, in such a manner that idle time is minimised and profit is maximised. To achieve this, the manager has to schedule customer requests on the basis of available resources, free machines and the cost and time of performing each task. Typical customer requests are of the form "produce item *u* given *v* by time *w* at cost *x*".

To aid the manager's scheduling process a diary is maintained for its current commitments. However, different applications will plan ahead for different periods of time, and be able to handle a different number of current tasks. Hence the first activity involves setting the planning ability of the agent.

DEF-1: Configuring Planning Parameters

This activity involves configuring the two parameters of the agent's internal Planner and Scheduler, these are:

- **Maximum Number of Simultaneous Tasks** - enter into this field the number of tasks the agent can perform concurrently, it is akin to the number of independent production lines in our metaphorical factory. The default value is 1 i.e. the agent can only attempts one task at a time. Whether or not this value needs to be changed is totally dependent on the role the agent will fulfil.
- **Planner Length** - enter into this field the number of time-grains that the agent will normally plan its activities, it is akin to the longest duration over which our metaphorical factory manager will books requests. The default value is 20, but the most appropriate value for any given agent will depend on the duration chosen for the time-grain, and the policy of the agent in question. For instance, if an agent can only plan ahead for a short period it may be unable to undertake long-term commitments, but may be more reactive to changing circumstances.

Note: unlike most other ZEUS agent attributes these parameters can not be changed at run-time.

See also…	**For information on…**
This Document, Section 3	Factors influencing the choice of time-grain
Technical Manual, Section x	How these parameters affect the operation of the Planner/Scheduler

DEF-2: Task Identification

This activity involves naming (but not yet actually defining) the application-specific tasks that this agent is capable of performing. This process by which this information is entered is illustrated in Figure 3.2.

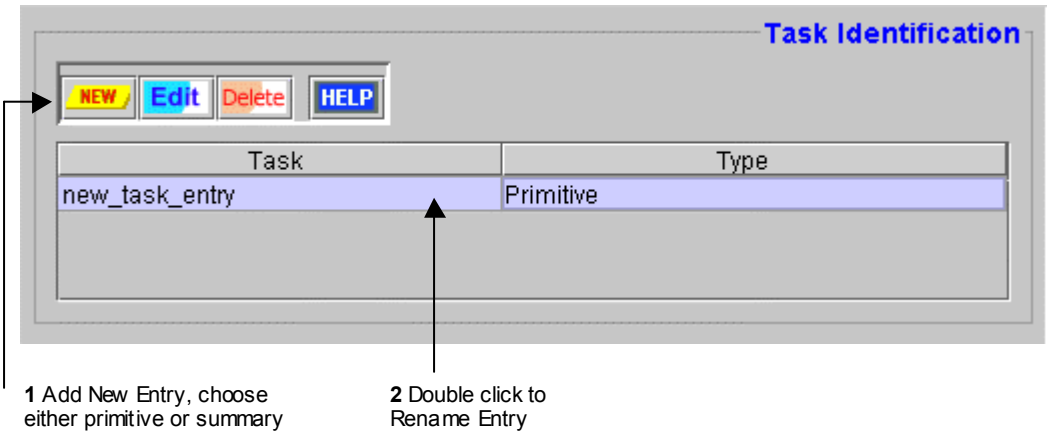


Figure 3.2: Screenshot of the Task Identification Panel, and how to use it

- Clicking on the toolbar's `New` button provides the choice of entering either a new Primitive task or new Summary task, or choosing a task that has already been entered.

A new entry will then appear in the table using a temporary name. (You may also notice that the newly created fact will also appear in the Task Table in the Generator's main window).

- To rename the task to something more meaningful, double click on its name field, edit the name and then press <return>.

See also…	For information on…
Technical Manual, Section x	The Characteristics of Primitive and Summary Tasks
This Document, Section 3.2	The Task Definition Process

DEF-3: Initial Agent Resource Allocation

This activity involves listing the resources that the agent will possess when it is initialised. This activity takes place through the 'Initial Resources' panel where the upper table lists the facts owned, and the lower table enables the facts' attributes to be edited, as shown in Figure 3.3.

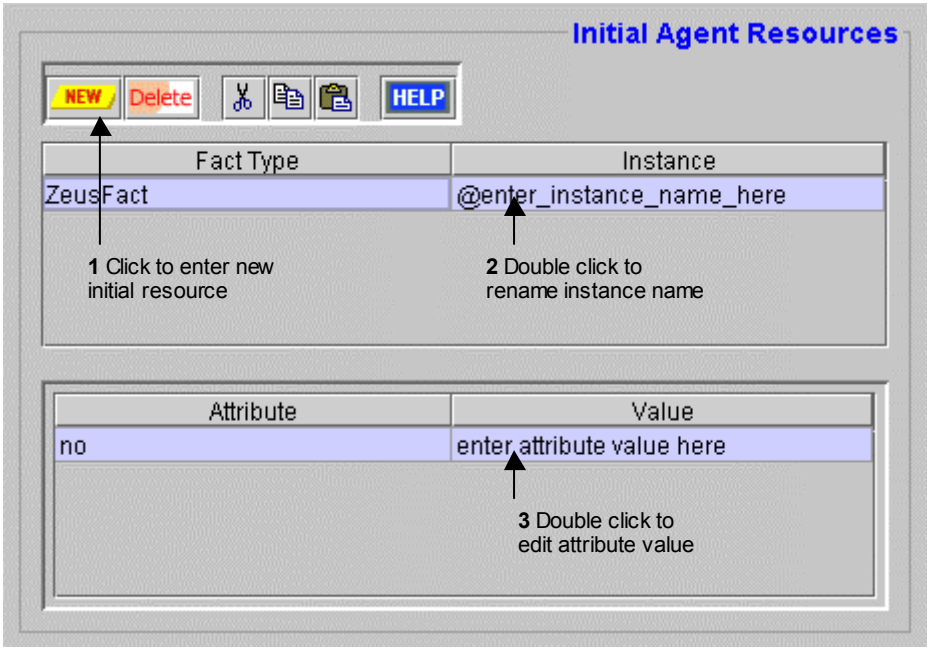


Figure 3.3: The Initial Agent Resources panel, and how to use it

Naturally, before a resource can be allocated to an agent it must first have been defined as part of the application ontology, (this should have been done during the previous stage).

- To enter a new initial resource click on the toolbar's `New` button, this opens a window showing the fact hierarchy of the project ontology. Select the appropriate fact, and then press the OK button.

- A new entry will now appear in table, consisting of a Fact Type (which can not be changed), and an Instance field that can be double-clicked to enter a more appropriate name.

Note that the @ character prefix indicates the resource is an individual instance, (and not any instance, which would be prefixed by a ? character).

- Because initial resources are individual instances it will probably be necessary to individualise their attributes. Achieve this by selecting the fact in question in the upper table and then double-clicking on the Value cells of the lower table to open a panel where a new value can be entered.
- Press <return> to enter the value and update the table. If the entered value appears in red then it is either syntactically incorrect or of an incompatible type, and will need to be changed.

Note: Whenever attribute values are being edited and another attribute needs to be referenced, right-click and a window showing the ontology's fact/attribute hierarchy will appear - just select the appropriate attribute and click OK.

3.2 The Task Definition Stage

By the time this activity is attempted the developer should have specified the list of tasks that the agent is capable of performing (see Activity **DEF-2**). Likewise, as the preconditions and effects of tasks are domain ontology concepts, the ontology should be comprehensively defined by the time this stage is attempted. If not, the developer may need to revisit the ontology devising process, although by this point the developer may have more focus and thus be better able to identify the appropriate application concepts.

- The tool for this stage is the Task Editor, which can be launched by double clicking on the task name in the Generator's Task Table, or selecting its entry in the and clicking the `Edit` button.

This will launch either the Primitive Task Editor (described in activity **TASK-1**) or the Summary Task Editor (described in **TASK-2**), depending on the underlying type of task.

TASK-1: Using the Primitive Task Editor

In ZEUS, a primitive task is a representation of some atomic (non-divisible) agent activity. As can be seen in Figure 3.4 in the Primitive Task Editor (PTE) depicts each task as a resource flow, where facts flow into a task, whereupon they are transformed into new facts.

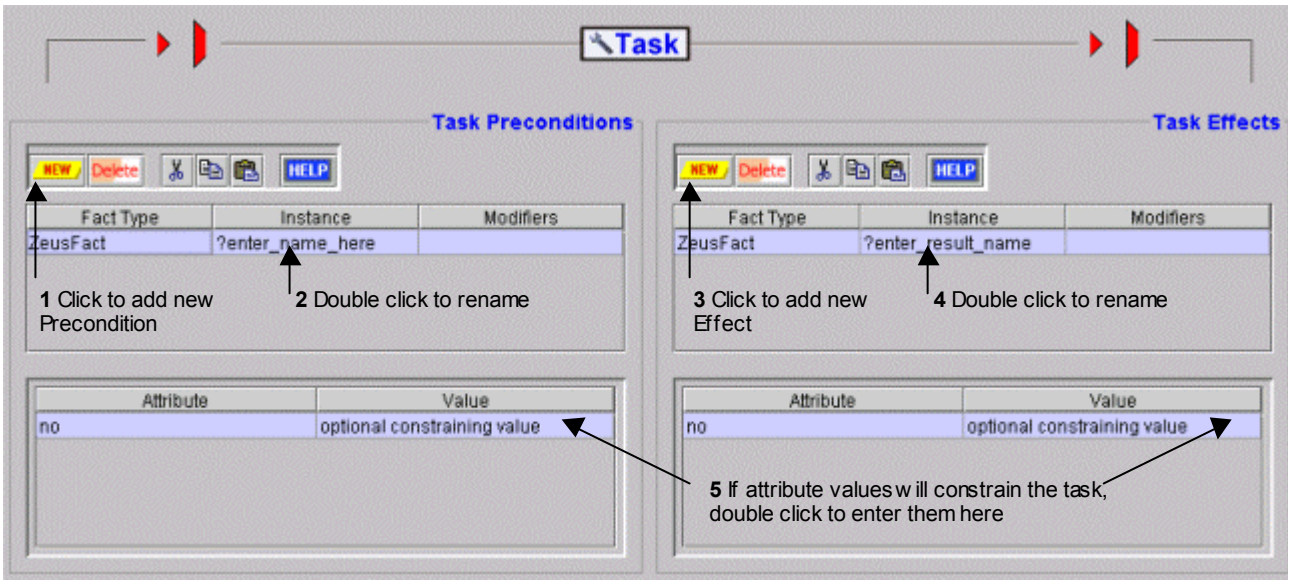


Figure 3.4: A Screenshot of the Primitive Task Editor with the main activities of the Task Definition Stage shown

Using the PTE, the following aspects of a primitive task can be configured:

- Preconditions - the resources needed for execution of the task
- Effects - the resources that will be produced upon execution of the task
- Cost - an expression given the cost of executing the task
- Duration - an expression giving the time taken to execute the task
- Precondition Ordering - the sequence in which preconditions should be achieved
- Constraints - applicability restrictions on executing the task

As some of these aspects are dependent on each other, their definition is best attempted in the order listed above.

How to Enter a Task Precondition

A precondition is a fact that will be used (and possibly consumed) by a task when it executes. Preconditions are listed in the top table in the 'Task Preconditions' panel, whilst their attributes are listed in the bottom table.

- To add a new precondition click on the toolbar's `New` button, this will open a pop-up window listing all the facts in the currently loaded ontology. Choose the appropriate fact and then click the OK button, the selected fact will then appear in the precondition table.

The instance name provides a handle for fact instances, akin to a variable name in programming languages. This can be edited and changed to something more meaningful by double clicking on its name. All names will be prefixed by the ? character to indicate that they are variables (i.e. any instance of that fact), rather than a specific instance.

- The Modifiers field describes how each precondition will be used. To set a modifier double click on this field, this will open a window listing the available options - click on the check-box beside the one that is appropriate.

The modifiers applicable to preconditions are described in table 3.1.

Modifier	Explanation	Examples of Use
Not	Task only performed if fact is not present in locally	For inhibitory factors, e.g. a produced fact that will inhibit production until it is removed
Is Read Only	Fact is not consumed, i.e. it survives task execution, but during which time it is not allocated exclusively to the task	For items of information, e.g. an employee record
Must be in Local Database	Fact must be in the agent's own local possession, prevents agent obtaining it from another agent	For items that should not be bought in, e.g. drive to work if a car is present, but don't buy one
Is Replaced after Use	Fact will be allocated to the task for the duration of its execution, but will not be consumed	For persistent entities capable of serving one task at a time, e.g. employees or vehicles

Table 3.1: The Precondition Modifiers

By default none of these modifiers are selected, it is assumed that all preconditions are reserved for the task's exclusive use, whereupon they will be consumed. The other options in the pop-up window: 'Is a Variable' and 'Is a Side-Effect only' are ghosted out, this is because preconditions are always variables, and never side effects.

How to Enter a Task Effect

An effect is a fact that represents the result of a task execution. The effects are listed in the top table in the 'Task Effects' panel, whilst their attributes are listed in the bottom table.

- New effects are added by clicking on the toolbar's `New` button, this opens a pop-up window listing all the facts in the currently loaded ontology. Choose the appropriate fact and then click the OK button, the selected fact will then appear in the effects table.

The name field provides a handle for particular groups of effects, akin to a variable name in programming languages. This can be edited and changed to something more meaningful by double clicking on its name. All names will be prefixed by the ? character to indicate that they are variables (i.e. any instance of that fact), rather than a specific instance.

- The Modifiers field for effects has a single option: 'Is a Side-Effect Only', is this is selected this task will not be selected by the Planner/Scheduler seeking to achieve this effect. Double click on the Modifiers cell of an effect to change this setting.

The difference between an effect and a side-effect is subtle; for instance, "lying on the ground" is a side-effect of hitting someone, but if the intention is to merely to instruct someone to lie on the ground, hitting them is a far from ideal way of achieving it. By default effects are not marked as side effects.

How to Enter the Task Cost

Each task can be given a numeric expression that approximates the cost of invoking it. The interpretation of this cost, and the units in which it is expressed are application specific.

- The cost is should be typed into the 'Cost' field of the lower 'Task Cost and Time' panel, then press <return> to commit the changes.
- To refer to another attribute in a previously entered precondition or effect, right-click to launch a window showing the ontology's fact/attribute hierarchy, then select the appropriate attribute and click OK, and it will appear inside the panel.

The default cost value is 0, (i.e. cost is not significant). This only needs to be changed if the `Planner/Scheduler` is to base its decisions on the cost of invoking tasks.

How to Enter the Task Duration

Each task can be given a numeric expression that approximates the length of time taken for it to execute. For consistency the duration should be interpreted in terms of time-grain units.

- The duration is should be typed into the 'Time' field of the lower 'Task Cost and Time' panel, then press <return> to commit the changes.
- To refer to another attribute in a previously entered precondition or effect, right-click to launch a window showing the ontology's fact/attribute hierarchy, then select the appropriate attribute and click OK, and it will appear inside the panel.

The default duration is 1 time-grain, this is also the minimum duration - even for tasks that are 'instantaneous'. Whether this value needs to be changed depends on the nature of the task concerned.

How to Constrain Task Attributes

The lower tables in the 'Task Preconditions' and 'Task Effects' panels enable the applicability of the task to be restricted. By default the value field of each attribute is empty, meaning that any variable of that type can be used to satisfy the task's precondition. If however a value is entered into one these fields, the task will only be executed if a fact with the appropriate attribute can be secured.

- To enter an attribute constraint, select the fact concerned from the upper table and then double-click on the Value cell of the relevant attribute in the lower table.
- This will open a panel where a new value can be entered, press <return> to enter the value and update the table. If the entered value appears in red then it is either syntactically incorrect or of an incompatible type, and will need to be changed.
- To refer to another attribute in a previously entered precondition or effect, right-click to launch a window showing the ontology's fact/attribute hierarchy, then select the appropriate attribute and click OK, and it will appear inside the panel.

Attribute constraints can either be literal values, or expressions can be evaluated into a value - this value will be compared against the corresponding attribute value of candidate facts at run-time to determine whether the fact is suitable for use.

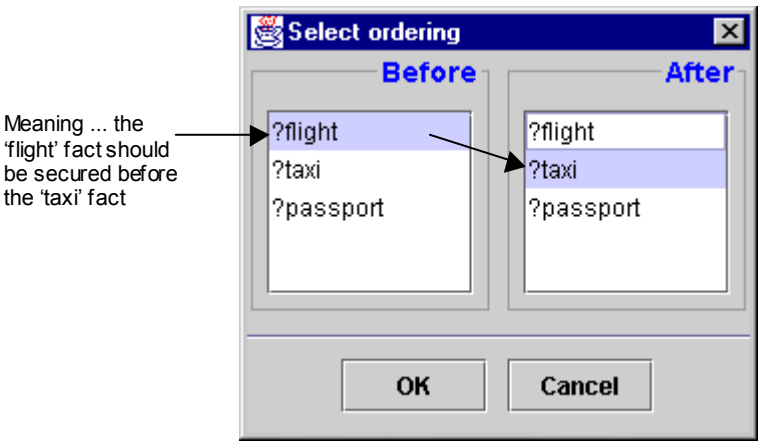
See also…	**For information on…**
Technical Manual, Section x	The Constraint Satisfaction Process

How to Order Preconditions

The precondition order of a task is a partial ordering of its preconditions, constraining the sequence in which they must be achieved. By default it is assumed that they can be considered in any order, however, sometimes certain resources should be considered before others.

This is useful in cases where the domain realisation of one precondition determines the acceptability of others. For example, in a planning a journey, there may be preconditions that a flight is booked and transportation to the airport arranged. Here we might want to state the flight should be booked first, since arranging transportation to the airport requires foreknowledge of the airport from which the flight departs.

- To specify an ordering, move to the PTE's 'Constraints' pane, it will be entered through its upper pane, the one labelled 'Preconditions Ordering Constraints'.
- Click on the toolbar's `New` button, this opens an ordering window consisting of two panels. Select the fact that should be secured first from the 'Before' column, and then select the fact it precedes from the 'After' column, i.e.: NEW



- As each ordering consists of a (before, after) pair, repeat as necessary for each precondition whose ordering is significant.

How to Enter Task Constraints

The 'Task Applicability Constraints' table is intended to enable the entry of constraints that do not relate to any particular fact (see "How to Constrain Task Attributes"). For instance, such a constraint might be 'do not attempt this task if the agent is unable to connect to the Internet'. NEW

- To enter one of these constraints click on this panel's `New` button, this will create a new entry in the constraints table.
- Then double click on the new entry to edit it, the constraint should be added in the form of an expression that will evaluate to true or false.

See also…	For information on…
Technical Manual, Section x	The Constraint Satisfaction Process

TASK-2: Using the Summary Task Editor

In ZEUS, a summary task consists of a number of primitive subtasks that need to be performed in some order to achieve its effects. Thus summary tasks are effectively mini-plans, which are useful for representational and cognitive economy (and planning efficiency). Summary Tasks are entered through their own editor, which is shown in Figure 3.5.

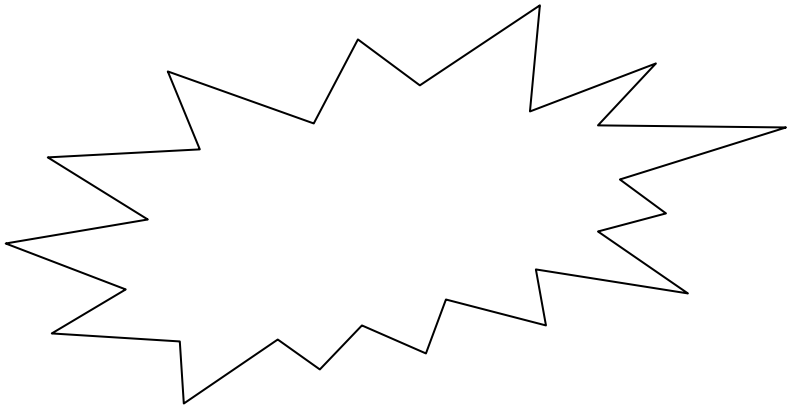


Figure 3.5: A Screenshot of the Summary Task Editor (STE)

Summary tasks are described in terms of a number of nodes and effect-precondition (producer-consumer) links between the nodes. Just as with primitive tasks, each node is defined in terms of its preconditions and effects. However, each node is simply a placeholder that can be replaced by any primitive or summary task with matching preconditions and effects. Like primitive tasks, summary tasks also have associated duration, cost and constraints; however, they lack a reference to an execution function since they cannot be executed directly.

This will be documented in a future release.

3.3 The Agent Organisation Stage

By the time this activity is attempted the application's agents should already have been defined, along with the tasks they are capable of performing. This stage involves supplying this knowledge about agents and abilities to the agents themselves, using the 'Agent Organisation' pane of the Agent Editor.

By default, agents are ignorant of names and the abilities of their neighbours, so if an agent needs the service of another it will need to contact a directory service to discover it. However, agents may have prior knowledge of other agents, especially if they interact with them on a regular basis. These known agents are called 'acquaintances'. There are four different types of relationships that can exist between agents, if they are acquainted they can be superiors, subordinates or co-workers, otherwise they will be peers. These relationships are described in Table 3.2.

Relationship	Explanation
Peer	The default relationship with no assumptions about agent interaction
Superior	The acquaintance is possesses higher authority than this agent, and can issue orders that this agent must obey
Subordinate	The acquaintance has less authority than this agent, and can be issued orders that it must obey
Co-worker	The acquaintance belongs to the same 'community' as this agent, and will be asked before peers when any resources are required

Table 3.2: The Acquaintance Relationship Types

Entering acquaintances and their relationships is one of the two activities of this stage, namely:

⇒ Entering Known Acquaintances, see activity **ORG-1**

⇒ Entering Known Abilities, see activity **ORG-2**

ORGANISATION ACTIVITY 1

ORG-1: Entering Known Acquaintances

Acquaintances are specified through the 'Known Acquaintances' panel, as illustrated in Figure 3.6.

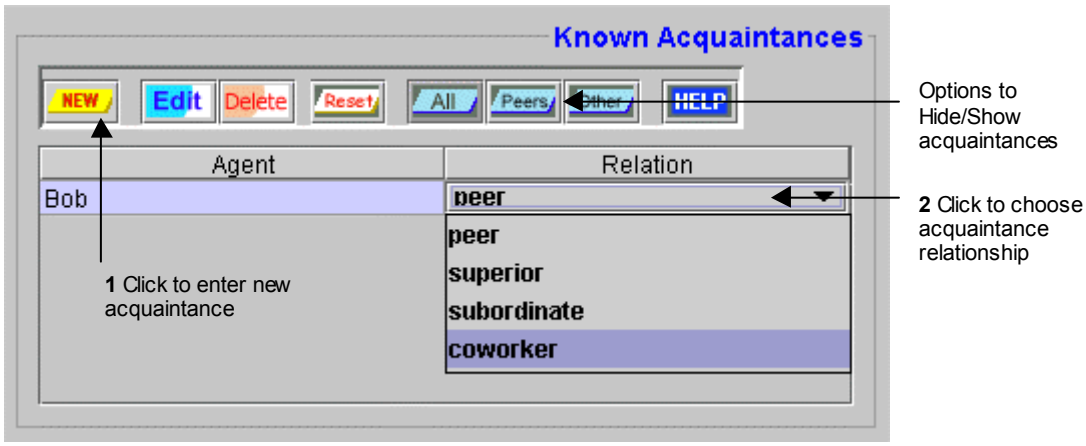


Figure 3.6: The Known Acquaintances panel, showing how to enter an acquaintance

- To enter a new acquaintance click on the toolbar's `New` button, this displays a list of all created so far in the current project, (with the exception of the agent being defined, of course).

- Select the agent that will be the acquaintance. (If the acquaintance has not been created yet select the 'Create New Acquaintance' option: this duplicates the function of the 'Add New Agent' button of the Agent Generator window, so rename the new agent as appropriate).
- A new entry for this agent will now appear in table with the relation field set to 'peer', which is the default value. To change this, double-click on the field, this will display a list of the possible relationship types, choose the one which is most appropriate.

Note: Organisational relationships are not bi-directional by default; i.e. there is no need for the agents' beliefs about their relationships with each other to be consistent.

- If an agent has many acquaintances, entries in the table can be filtered by clicking on the 'All', 'Peers' and 'Other' buttons: these will show only those acquaintances with that relationship.

- If, at a later date, you want to remove the authority and co-worker relationships from an agent, click on the '`Reset`' button, this will retain all the acquaintances, but set their relationship with the current agent to 'peer'.

Once acquaintances have been specified, you may want to specify the abilities that this agent knows they can perform, this is achieved during the next activity.

ORG-2: Entering Known Abilities

Whereas the previous activity dealt with the identities and relationships of acquaintances, this activity deals with their abilities. This is particularly useful for setting up preferred supply relationships, since if an agent knows an acquaintance can provide a resource it contact it first before asking the directory service to recommend a supplier. Abilities are specified through the 'Known Acquaintance Abilities' panel, as illustrated in Figure 3.7.

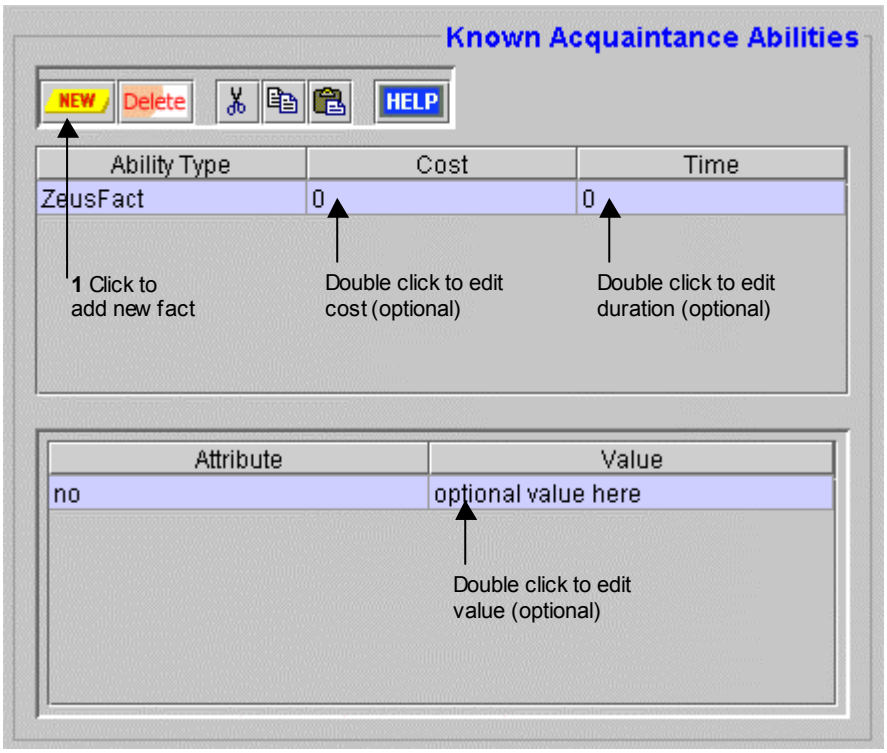


Figure 3.7: The Known Abilities Panel, showing how to enter an acquaintance's ability

- As abilities are associated with acquaintances you must first choose the acquaintance that will possess the ability: do this by clicking on the appropriate entry in the 'Known Acquaintances' table, (the table shown in Figure 3.6).
- You can now enter abilities for the selected acquaintance by clicking on the `New` button. This opens a window listing all the facts in the currently loaded ontology. Choose the appropriate fact, click the OK button and an entry will appear in the table.

- If the agent has any prior knowledge of the cost of invoking this ability, this can be entered into the ability's 'Cost' field.

This is analogous to the part of the Task Definition stage where the cost of performing the task could be estimated. Of course, as this value is an estimate its primary purpose is to allow the agent to rank potential suppliers. In practice the price charged by suppliers may vary depending on circumstances. The default value is 0, in which case the agent will not have any preconceptions about price.

- If the agent has any prior knowledge about the duration of this ability, this can be entered into the ability's 'Time' field.

Again, this value only serves an estimate allowing the agent to rank potential suppliers, and in practice the time taken to complete the task may vary depending on circumstances. The default value is 0, in which case the agent will not have any preconceptions about the length of the task.

- When an ability entry is selected in the table its attributes are shown in the table below. The values of these attributes can be edited to reflect the nature of production. For instance, if only one object is produced by this task's invocation 1 should be entered in the 'number' field.

3.4 The Agent Co-ordination Stage

This, the final stage of the Agent Creation process, involves equipping the agent with the co-ordination protocols and expertise required for social interaction with other agents. This information is entered through the 'Agent Co-ordination' pane of the Agent Editor.

An Insight into Agent Interactions

But before attempting this stage the developer should have a clear idea of how the agents will interact in the course of fulfilling their roles. All ZEUS agent interactions are variations on the multi-round contract-net, [x], which involves one or more *Initiators* that issue a call for proposals (CFP), and one or more *Respondents* that reply. If you are unsure whether a particular role is an Initiator or a Respondent consult the appropriate role model for guidance.

The key aspects of any agent interaction are the co-ordination protocol and the negotiation strategies. The co-ordination protocol is an agent conversation model that describes when each party is expected to communicate, what messages will be exchanged and the effect of receiving particular messages. As the states assumed by the Initiator and Respondent roles are different, the default co-ordination protocol is described from both perspectives, as shown in Figure 3.8.

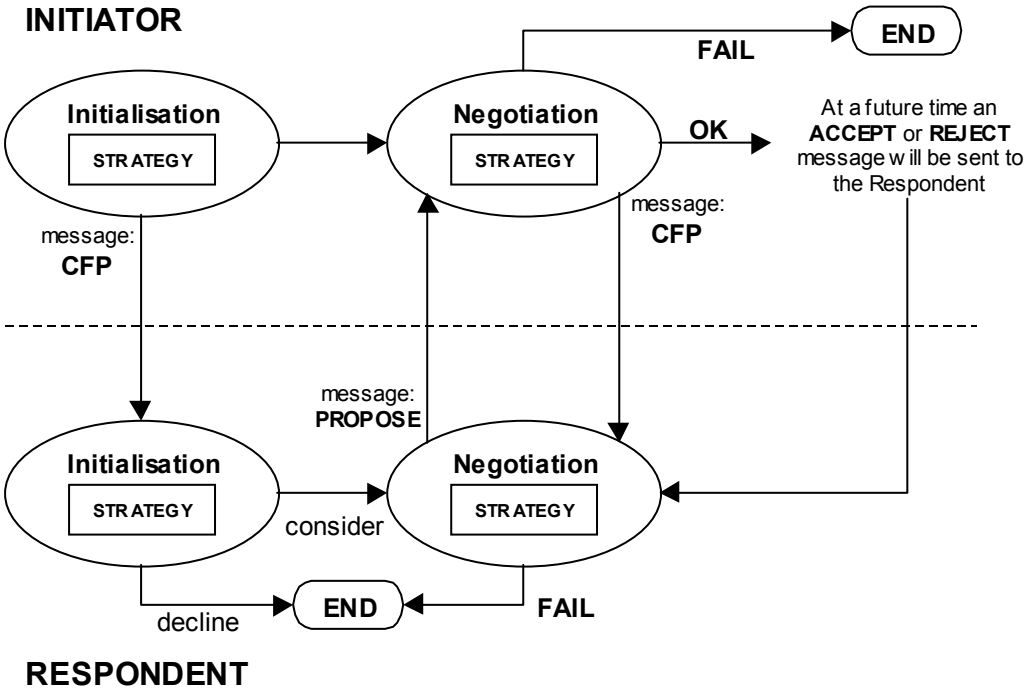


Figure 3.8: A state-transition diagram of a typical contract net negotiation

Figure 3.8 also illustrates the relationship between protocols and interaction strategies. At each state the agent may need to make decisions about how to behave or respond to its current circumstances, these decisions are decided by strategies. This is best illustrated with an example.

Consider an agent that needs to obtain a resource, because it can not produce it locally it must contact another agent to supply it. Hence the Initiator begins in the *Initialisation* state by analysing its requirements and determining how much it is willing to pay for the resource and how quickly it needs it. Using the expertise encoded into its tendering strategy it formulates a **CFP** message containing its requirements, this is then broadcast to all potentially interested parties and the agent moves into the *Negotiation* state to await responses.

The arrival of a CFP message causes the Respondent to move into its *Initialisation* state. If the Respondent decides to reply (it is under no obligation to do so) it will move into its *Negotiation* state. The Respondent will now use its evaluation strategy to formulate a counter-proposal to the initial tender, which is then sent back to the Initiator in the form of a **propose** message. The Respondent then moves into a wait state for a finite period of time to await a response, no response is assumed however, and if nothing is received by the end of its time-out period the agent will terminate its part of the conversation.

When the Initiator receives a proposal it is analysed using its own evaluation strategy, this can have three results:

- If the proposal is acceptable the conversation ends. The Initiator has not committed itself however, and will send a message at some point in the future either accepting or rejecting the offer.
- If the proposal is not acceptable and the Initiator decides there is little point in negotiating further it can end the conversation immediately.
- If the proposal is not acceptable the Initiator can send a new, modified CFP message to the Respondent in question, whereupon it enters a wait state until a response arrives or its time-out period passes.

In the latter case, the Respondent is awoken by the arrival of a new CFP message, which is analysed using its local evaluation strategy. This will cause the Respondent to do one of the following:

- It will decide not to bid again and end its side of the conversation.
- It will formulate a new proposal message, return it to the Initiator, and move into a wait state until a response arrives or it times out.

Having seen how agents interact, it is clear that the key factors are the protocol and interaction strategy. Hence the purpose of the Agent Co-ordination stage is to both specify which aspects of the contract net protocol will be available to the agent, and to choose the strategy that will determine the agent's behaviour during interaction. This is reflected in the three activities that can be performed during this stage, namely:

⇒ Equipping the agent with a co-ordination protocol, see activity **COORD-1**

⇒ Equipping the agent with an interaction strategy, see activity **COORD-2**

⇒ Adding new Interaction Protocols and Strategies, see activity **COORD-3**

COORD-1: How to Equip an Agent with a Co-ordination Protocol

The ZEUS tool-kit provides pre-built co-ordination protocols that implement various aspects of contract-net type conversations, and which are applicable for a variety of agent applications. The available protocols are listed in the 'Co-ordination Protocols' panel, as shown in Figure 3.9.

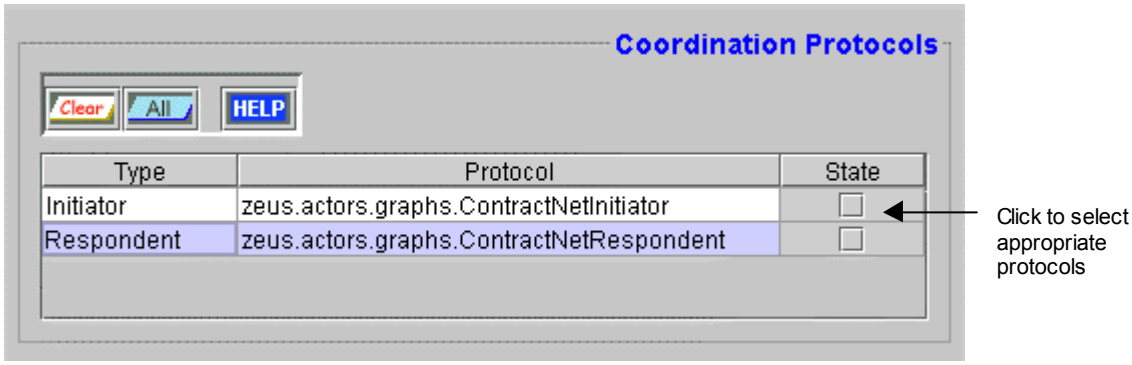


Figure 3.9: The Co-ordination Protocol entry table, showing two typical ZEUS protocols

In Figure 3.9 the 'Protocol' field refers to the Java file within the ZEUS class library that provides the implementation of that protocol. The 'Type' field refers to the nature of the protocol, which in keeping with the model of interaction presented earlier, will be either *Initiator* or *Respondent*.

- To equip the agent with a protocol just click on the check-box beside its name, a tick will appear to indicate its selection, clicking again will clear the selection.
- To select all the protocols, click on the 'All' button in the toolbar. All the protocols can be deselected by clicking on the 'Reset' button.

- Once a protocol has been selected an entry will appear in the 'Co-ordination Strategies' table; (this is further explained in the entry for activity **COORD-2**).

See also…	**For information on…**
Technical Manual, Section x	Descriptions of the ZEUS Co-ordination Protocols
Technical Manual, Section x	How the Co-ordination Engine uses Protocols

COORD-2: How to Equip an Agent with an Interaction Strategy

Whereas interaction protocols describe how agents interact, it does not explain why. This arises from each agent's intentions and motivations, which are encoded into interaction strategies. Agent developers can choose an appropriate strategy from those supplied with the ZEUS toolkit or create their own, (see activity **COORD-3**).

Interaction strategies refer to the protocol currently selected in Figure 3.9, and are entered through the 'Co-ordination Strategies' panel, as shown in Figure 3.10.

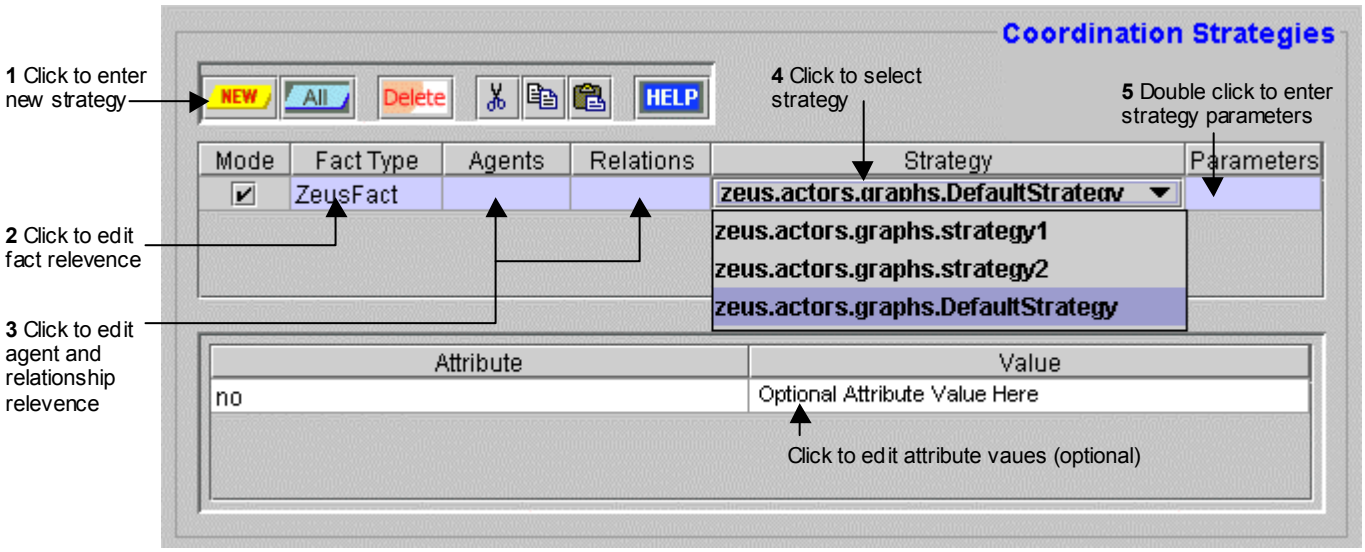


Figure 3.10: The Co-ordination Strategy entry panel, and how to use it

Basically, the objective of this activity is to specify what strategies with what parameters are used to obtain particular facts from particular agents.

- Before entering a strategy ensure that the protocol to which it refers is the currently selected entry in the 'Co-ordination Protocols' table (the table in Figure 3.9).

On selecting a protocol a single entry will appear in the Strategies table. As the values of the fields suggest, this default strategy can be used to obtain any type of fact from any agent regardless of its relationship.

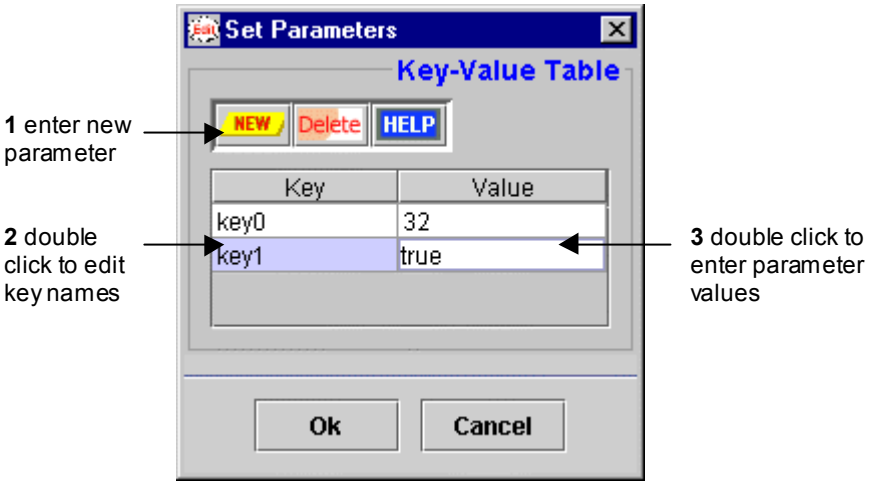
- To add a new strategy click on the toolbar's `New` button, this will create a new entry in the strategies table.

- The default setting for the `Mode` field is checked, indicating the strategy will be chosen when negotiating for the fact type in question. If the opposite is true and this strategy will never be used to obtain that fact, click on this field to deselect it.
- By default the new entry refers to obtaining facts of any type, (which is why the `Fact Type` field is set is to 'ZeusFact', the parent class of all facts). If the strategy is more specific, click on this field, this opens a fact hierarchy window enabling a more appropriate fact type to be chosen.
- Specifying the values of the expected attributes in the lower table can further refine the fact obtained by this strategy. By default no values are associated with the fact attributes, meaning that the strategy can be used to obtain any fact of that type. If this is not the case, double click on the appropriate `Value` field to restrict the strategy to facts that contain that value.
- If the agent has acquaintances, the strategy can be modified so that it is only used in for negotiations with named agents by double clicking on the `Agents` field, and then choosing them from the list. This field is empty by default, signifying that it can be used with any agent.
- A strategy can also be modified so that it is only used in conjunction with acquaintances that have a certain relationship to this agent. Specify this by double clicking on the `Relations` field, and

choosing the appropriate relationships. By default this field is empty, meaning the strategy will be used regardless of any relationships that exist between agents.

A strategy can be specified as being applicable to only certain agents *and* certain relationships. An example would be interpreted as 'use this strategy only in conjunction with agent A if and only if you are A's co-worker.

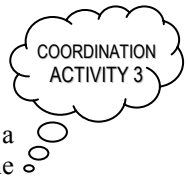
- The strategy itself is a body of procedural knowledge that will have been implemented as a Java file and incorporated into one of the ZEUS packages, (see activity **COORD-3**). By default new strategy entries will refer to zeus.actors.graphs.DefaultStrategy, to select another double click on the `Strategy` field, this shows a list of all currently loaded strategies, whereupon you can choose the one most appropriate.
- The procedural nature of strategies means that they behaviour can be modified by passing different parameters. By default no parameters are specified and so the `Parameters` field is empty; but if they are required, double clicking the field opens a window like this:



Currently the parameter entry window does not automatically parse the selected strategy to identify the parameter names and their types, placing the onus on the developer to refer back to the Java implementation of the selected strategy to identify which parameters are required. The parameters will be passed into the strategy as a Java Hashtable object, hence the `Key` and `Value` fields map directly to the key and value parts of Hashtable entries. Note that no syntax checking is performed upon entry, instead the values are passed to the strategy as strings and parsed there.

- To enter a new key, press the `New` button, this creates a new key-value entry. NEW
- Double click on the `Key` field to rename it to the identifier that the strategy expects to find in the Hashtable. Then double click on the `Value` field and enter the appropriate value.

See also…	For information on…
Technical Manual, Section x	Descriptions of the ZEUS Co-ordination Strategies
Technical Manual, Section x	How the Co-ordination Engine uses Strategies



COORD-3: How to add new Interaction Protocols and Strategies

The interaction strategies used by agents are very dependent on the application being developed, as a consequence those supplied with the ZEUS toolkit may not be appropriate. To solve this problem the ZEUS toolkit provides a framework around which new strategies can be created and integrated.

Currently adding a new strategy is not supported from within the ZEUS Agent Generator tool, (although support may be added in future releases); thus to add a strategy you should do the following:

- Save your current project and quit from the Generator tool.

You will now need to edit the ZEUS toolkit properties file, the name and location differ slightly depending on what operating system you are using.

- If you are using Windows NT, look in the directory that holds your user profile, (this is typically `c:/winnt/profiles/<your username>/)`. You should see a file called `zeus.prp` - open it in your text editor.
- If you are using Unix or its variants, you will need to edit the file called `.zeus.prp`; you will find it in your root directory (just use the ~ shortcut).

Once you have opened the ZEUS properties file you can add edit to include the new strategy. This process is analogous to the way in which system variables are set, consequently entries in the properties file must be separated by colons, (i.e. entry1:entry2:entry3).

Warning: If you alter the properties file you must adhere to the correct syntax, otherwise you will not be able to restart the Agent Generator. Consequently it might be a good idea to create a back-up copy of the properties file before editing it.

- If you are adding a new Initiator protocol, locate the line that begins "`user.protocols.initiator="` and add the **full pathname** of the class that implements the protocol, (not forgetting to separate it from any existing entries by colons).

  So, if you wanted to add the protocol called `myWay` implemented in a file called `myWay.java` in the `zeus.test.myapp` directory, you would add the entry: `zeus.test.myapp.myWay`

- If you are adding a new Respondent protocol, locate the line that begins "`user.protocols.respondent="` and add the **full pathname** of the new protocol.

- If you are adding a new Initiator strategy, locate the line that begins "`user.strategy.initiator="` and add the **full pathname** of the new strategy.

- If you are adding a new Respondent strategy, locate the line that begins "`user.strategy.respondent="` and add the **full pathname** of the new strategy.

Once the new strategy entry has been added, you should save the properties file and restart the Agent Generator. If you go to the Agent Co-ordination panel you should find that the new protocols or strategies are present in the respective tables (the ones illustrated in Figure 3.9 and 3.10).

Obviously, this activity only makes the new strategy known to the Agent Generator. Before the application can use the new strategy it must be implemented, this process is described in activity **IMPL-5** in Section 6.

4 THE UTILITY AGENT CONFIGURATION STAGE

By now the task-specific agents will have been defined, leaving us to consider the agents that will provide the support infrastructure: these are known as the utility agents. This stage requires that the following design decisions should have already been taken:

- how the name resolution service will be realised
- whether there will be a Facilitator, and if so, what are its attributes
- whether there will be a Visualiser
- whether agent activity will be stored persistently

The number and nature of the utility agents needed is dependent on the application. The Role Modelling guide and its case studies provide examples of how to configure utility agents appropriately. Each type of utility agent can be configured through the 'Utility Agent' pane of the Code Generator window. To open the Code Generator tool select the "Generate Code" menu option or the equivalent button from the Project Options toolbar. You will now be able to perform the main activities of this stage, namely:

⇒ Configuring the Name Servers, see activity **UTIL-1**

⇒ Configuring the Facilitators, see activity **UTIL-2**

⇒ Configuring the Visualisers, see activity **UTIL-3**

⇒ Configuring the Database Proxies, see activity **UTIL-4**

UTIL-1: Configuring the Name Servers

An agent society must possess at least one Agent Name Server (ANS). The ANSs maintain a registry of known agents, enabling them to map agent identities to a logical network location. This is necessary because agents only know the names of their acquaintances and not their locations. Name Server agents are created and configured using the 'Name Servers' panel, as shown in Figure 4.1.

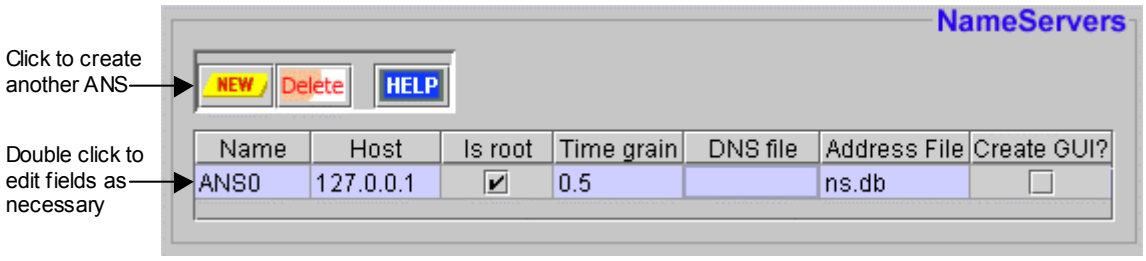


Figure 4.1: The Name Servers Entry Panel

By default the Name Servers table has one entry, referring to a single ANS agent, this should suffice for small agent societies. However as the ANS is vital to all agent communication it is a potential bottleneck, and so it may be desirable to have multiple ANSs to support larger societies, or to provide a degree of redundancy in case one fails.

- To add another ANS, click the `New` button, this will create another entry in the table that can be edited as required.

The `Host` field shows the I.P address (i.e. network location) of the machine that the ANS will run on, this defaults to the I.P address of the machine that is currently running the Generator tool. More than one ANS can reside on a single host.

- To change the location of the ANS double click on the `Host` field and enter the new value. If you don't know the I.P address of machine, use the ping command to discover it.

If there is more than one ANS the developer can choose which is the root server, the only operational difference of changing this is the root server will provide the time-grain value (see below) and be responsible for maintaining the society-wide clock.

- To select or deselect an agent as the root, click on the 'Is Root?' field. Note: one ANS must be selected as the root.

The agent selected as the root will provide the time-grain value for the entire society, (for a discussion of the time-grain and its implications see section 3 of this document).

- Change the time-grain by double clicking the `Time Grain` field, the new value should be expressed in minutes and be a non-negative real number. For instance to set the time-grain to 20 seconds 0.33 should be entered. Note: only the time grain field of the root ANS can be edited.

As the root ANS serves as a reference point for all other agents in the society, there must be a means to inform other agents where the root ANS is located. Hence when the root ANS starts, it will write its network location into a file called the Default Name Server (DNS) file. If agents share a network file system it is recommended that the DNS file be expressed in terms of a network pathname.

- To edit the pathname to which the root ANS will write the DNS, double click on the `Address File` field of the root agent.

To make non-root name servers aware of the root ANS, they must be told where to find the root's DNS file. This pathname is entered into the `DNS File` field. If this is not a network accessible file, it will need to be copied to the local file system of the agent concerned, and hence this field will contain its local pathname and filename.

- To edit the pathname from which the DNS file will be read, double click on the `DNS File` field of the root agent.

The developer can choose whether Agent Name Servers will be created with a graphical user interface, (GUI). The GUI will display information on the server's activity, and provide some control functions. This may be useful for debugging and monitoring, but unnecessary if the ANS is to run in the background.

- To create an ANS with a GUI click the `'Create GUI?'` checkbox field, click again to deselect.

UTIL-2: Configuring the Facilitators

Whereas every agent society must have an ANS, there is no such obligation for Facilitators. Whether Facilitators are included depends on the nature of the application; if all agent acquaintances and abilities have been determined at design-time and will not change, then a Facilitator may not be necessary. However, this situation is unlikely, and so most applications will possess at least one Facilitator, which can be created and configured using the 'Facilitators' panel, as shown in Figure 4.2.

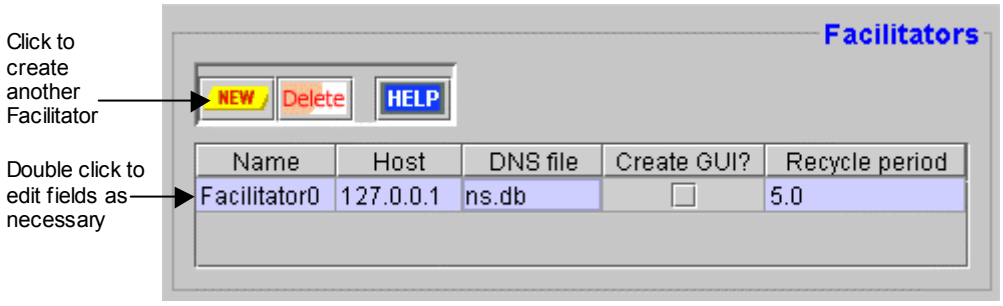


Figure 4.2: The Facilitator Entry Panel

By default the Facilitators table has one entry, one should suffice for small-scale applications, but as it could be a potential bottleneck it may be desirable to have multiple Facilitators for larger applications, or where some redundancy is wanted in case of failure.

- To add another Facilitator, click the `New` button, this will create another entry in the table that can be edited as required.

The `Host` field shows the I.P address (i.e. network location) of the machine that the Facilitator will run on, this defaults to the I.P address of the machine that is currently running the Generator tool. More than one Facilitator can reside on a single host.

- To change the location of a Facilitator double click on the `Host` field and enter the new value. If you don't know the I.P address of machine, use the ping command to discover it.

To make Facilitators aware of the root ANS, they must be told where to find its DNS file. This pathname is entered into the `DNS File` field. If this is not a network accessible file, it will need to be copied to the Facilitator's local file system, and its local pathname and filename entered in this field.

- To edit the pathname from which the DNS file will be read, double click on the `DNS File` field.

The developer can choose whether Facilitators will be created with a graphical user interface, (GUI). The GUI will display information on the agent's activity, and provide some control functions. This may be useful for debugging and monitoring, but unnecessary if the agent is to run in the background.

- To create a Facilitator GUI click the `'Create GUI?'` checkbox field, click again to deselect.

How to Make the Facilitator Reactive

The normal behaviour of a Facilitator is to retrieve the list of known agents from an ANS, and then send a message to each of the agents asking them to reply with details of the abilities they are currently able to perform. The interval between these queries is set through the `Recycle Period` field.

- Change the cycle interval by double clicking the `Recycle Period` field, the new value should be expressed in minutes and be a non-negative real number.

If the cycle period is set to 0 the Facilitator will cease its proactive querying of agents and become totally reactive. In this case the normal roles are reversed and the agents must behave proactively, periodically advertising their abilities to the Facilitator. This will require altering the agent's normal behaviour, (see activity **IMPL-4A**).

UTIL-3: Configuring the Visualisers

Like Facilitators, there is no requirement for an application to contain a Visualiser. Whether one is included depends on whether the application is to be debugged, monitored or analysed. Given the Visualiser offers some very useful functionality for free it will usually be included in the list of agents to be created. Visualisers are created and configured using the 'Visualisers' panel, shown in Figure 4.3.

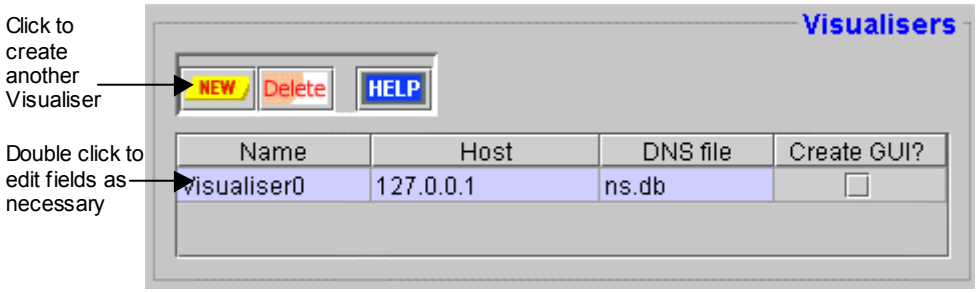


Figure 4.3: The Visualiser Entry Panel

By default the Visualisers table has one entry, whether more than one is necessary will probably depend on the number of locations where users will want to visualise some aspect of the society. Another influencing factor is that Visualisers are not essential to the operation of an agent society, and so as the implications of failure are less serious there is less need for redundancy.

- To add another Visualiser, click the `New` button, this will create another entry in the table that can be edited as required.

The `Host` field shows the I.P address (i.e. network location) of the machine that the Visualiser will run on, this defaults to the I.P address of the machine that is currently running the Generator tool. As the Visualiser windows will occupy most of a host's screen space, there is likely to be only one Visualiser per host (although this is not enforced).

- To change the location of a Visualiser double click on the `Host` field and enter the new value. If you don't know the I.P address of machine, use the ping command to discover it.

To make Visualisers aware of the root ANS, they must be told where to find its DNS file. This pathname is entered into the `DNS File` field. If this is not a network accessible file, it will need to be copied to the Visualiser's local file system, and its local pathname and filename entered in this field.

- To edit the pathname from which the DNS file will be read, double click on the `DNS File` field.

The developer can choose whether Visualisers will be created with a GUI that will display information on the Visualiser agent's activity; (this will be created separately from the windows of the Visualiser tools). This GUI may be useful for debugging and understanding how the agent works, but otherwise this option should be ignored.

- To create a Visualiser with an activity GUI click the `'Create GUI?'` checkbox field; click again to deselect.

UTIL-4: Configuring the Database Proxies

A Database Proxy (DP) agent provides Visualiser agents with a means of persistent storing agent session information. A DP can serve as an interface to a 3rd party database, or translate the information to be stored into its own ASCII file format. Database Proxies are created and configured through the panel of the same name, shown in Figure 4.4, although there is no entry there by default.

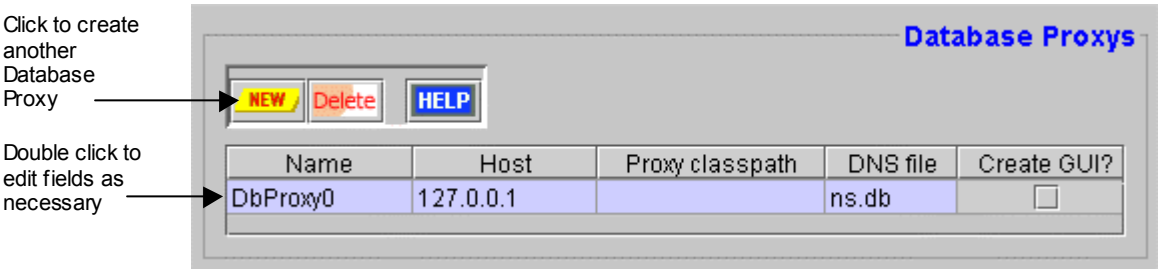


Figure 4.4: The Database Proxy Entry Panel

Whether a DP is included in an application depends on whether the activities of the agents need to be logged. Hence a DP will typically only be created if the society is being debugged or audited, and a DP will not be created by default.

- To add a Database Proxy, click the `New` button, this will create an entry in the table that can be edited as required.

The `Host` field shows the I.P address (i.e. network location) of the machine that the Database Proxy will run on, this defaults to the I.P address of the machine that is currently running the Generator tool. If the DP is to be connected to a proprietary database the address entered will probably need to be the address of the machine that hosts the database.

- To change the location of the DP double click on the `Host` field and enter the new value. If you don't know the I.P address of machine, use the ping command to discover it.

Like the other utility agents the DPs must be aware of the root ANS. Thus the location of the DNS file should be entered into the `DNS File` field. If this is not a network accessible file, it will need to be copied to the DP's local file system.

- To edit the pathname from which the DNS file will be read, double click on the `DNS File` field.

The developer can choose whether the DP will be created with a GUI that will display information on its activities. This GUI may be useful for debugging and understanding how the agent works, but otherwise this option should be ignored.

- To create an activity GUI click the `'Create GUI?'` checkbox field; click again to deselect.

How to Connect a Database Proxy to a Storage Medium

The Database Proxy is connected to its storage media through a Java class that implements the methods of the storage interface. The ZEUS toolkit provides two predefined storage interfaces:

- zeus.ext.FlatFile : this writes saved information into a conventional ASCII file
- zeus.ext.Oracle : this writes information into an Oracle Database

- To choose the storage mechanism to be used, double click on the `Proxy Classpath` field and enter the full pathname of the appropriate class. If this is not entered the DP will not be able to save information.

5 THE TASK AGENT CONFIGURATION STAGE

During the previous stage, utility agent configuration, the run-time parameters of the utility agents were specified. During this stage the same process is undertaken for the application's task agents. In keeping with previous stages, this also assumes certain design decisions have been taken:

- whether each agent will be linked to an external source of data
- whether each agent will be linked to external programs (like a GUI front-end)

For those seeking guidance, the Role Modelling guide and its case studies provide examples of typical task agent configurations. All task agents are configured through the 'Task Agents' pane of the Code Generator window. To open the Code Generator tool select the Agent Generator's "Generate Code" menu option or the equivalent button from the Project Options toolbar. This will enable the developer to perform the sole activity of this stage:

⇒ Configuring the Task Agents, see activity **TACF-1**

TACF-1: Configuring the Task Agents

Each task agents in the current project appears in the Code Generator's agent table (as shown in Figure 5.1). Shown beside its name is status, this will be either *Saved* - which indicates the agent has not been altered since the project was last loaded; or *Modified* - which indicates the agent has been modified recently and so its source code will need to be regenerated to take account of the changes.

The same status information is provided for tasks in the 'Tasks' pane, where you can also choose which tasks will be generated when the Code Generator is executed.

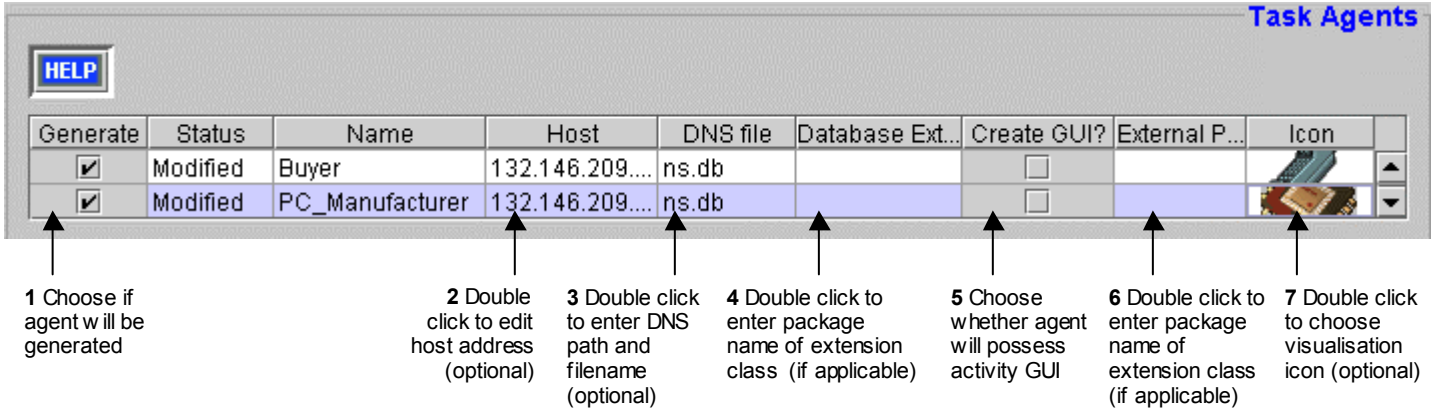


Figure 5.1: The Task Agent Configuration Panel, and how to use it

The `Host` field shows the I.P address (i.e. network location) of the machine that each task agent will run on. This defaults to the I.P address of the machine that is currently running the Generator tool. The machine resources consumed by a single agent will depend on the application, although from our experience it is perfectly reasonable to run many agents on the same host. However, agent load balancing and assessment is somewhat beyond the scope of this document.

- To change the location of a Visualiser double click on the `Host` field and enter the new value. If you don't know the I.P address of machine, use the ping command to discover it.

Like the utility agents, the task agents need to be aware of the root ANS, and must be told where to find its DNS file. Thus the location of the DNS file should be entered into the `DNS File` field. If this is not a network accessible file, it will need to be copied to each task agent's local file system.

- To edit the pathname from which the DNS file will be read, double click on the `DNS File` field.

The icon field is a cosmetic rather than a functional option; it enables the developer to choose what icon will represent the agent when the Visualiser displays it. As the Visualiser uses the icon it should be accessible from its file system - if not, it should be copied over to it. (Icons should be 256-colour GIF format images, preferably about 32 to 64 pixels in both width and breadth).

- To set a task agent's icon, double click on its `Icon` field, this will open a file dialog window enabling the icon file to be located and chosen.

The icon locations will be placed in a file called *default.gifs* - this should be located in the same directory as the name server file used by each Visualiser.

How to Link an Agent to an External Resource

A task agent can obtain the resources it needs to perform its tasks from one of three sources: from its resource database, from other agents, our from an external data source. The latter can be connected to the task agent through a Java class that implements the *zeus.actors.ExternalDb* interface.

- To connect the task agent to an external data source double click on the `Database External` field and enter the full package name of the appropriate class.

See also…	For information on…
This Document, Section 6	How to connect to external resources (activity **IMPL-3**)
Technical Manual, Section x	The ZEUS agent API

How to Link an Agent to an External Program

A task agent can be linked to external programs that enable it to send or receive information from the outside world. This external program will typically be a user interface from which instructions are received and to which the agent sends results. External programs are connected to the task agent through a Java class that implements the zeus.agents.ZeusExternal interface.

- To connect a task agent to an external program double click on the `External Program` field and enter the full package name of the appropriate class.

See also…	For information on…
This Document, Section 6	How to write external programs (activity **IMPL-4**)
Technical Manual, Section x	The ZEUS agent API

How to Create an Agent with an Agent Viewer GUI

The Agent Viewer GUI is a specialised tool that displays detailed information on the internal components of agent. This is independent of any other application front-end and provides the best means for understanding how the agent works, as well as being an excellent debugging aid. Whether a task agent is created with this GUI is left to the discretion of the developer.

- To create a task agent with an Agent Viewer GUI click the `'Create GUI?'` checkbox field; or click again to deselect.

See also…	For information on…
Agent Viewer User Manual	The Features of the Agent Viewer

6 THE AGENT IMPLEMENTATION STAGE

Once the task and utility agents have been appropriately configured the next stage is to generate the application source code. This process again involves using the Code Generator tool, which is launched by selecting the "Generate Code" option from the Agent Generator's menu bar, or the equivalent button from the Project Options toolbar. The following activities can now be attempted:

⇒ Generating the Task and Agent Source Code, see activity **IMPL-1**

⇒ Implementing the Task Bodies, see activity **IMPL-2**

⇒ Implementing the External Resources, see activity **IMPL-3**

⇒ Implementing the External Programs, see activity **IMPL-4**

⇒ Implementing a new Interaction Strategy, see activity **IMPL-5**

IMPL-1: How to Generate Task and Agent Source Code

As the information used to generate the agents is taken from the Generator's internal model, you should ensure that the project has been saved before attempting this stage. Code generation is achieved through the "Generation Plan" tab pane, as shown in Figure 6.1.

Unlike most other tables of the Agent Generator, the main table of Figure 6.1 can not be edited. Instead the function of this table is to display information on which agents and tasks will be created when the code generator tool is run next.

- To remove an agent or task from this list, go to the "Task Agents" or "Tasks" panes and uncheck the "Generate" checkbox beside the task or agent name.
- Alternatively select the entry to remove and press the toolbar's `Delete` button. Whilst selecting the `Clear` button at the top of the screen will remove all entries.

The table's type field provides an at-a-glance reference to the type of each entry; this will be Name Server, Facilitator, Visualiser, Agent or Task. Clicking on the button in the toolbar above the table enables particular entry types to be hidden or shown.

The next step is to decide where the source code will be written. As the generator will overwrite previously generated files in the same location this should be chosen carefully.

- To change the destination of the generated source code click on the `Change Target Directory` button. This will open a file dialog window, use this to navigate to the intended target directory. Note however that a 'feature' within the current Java file dialog means you will need to choose a file within the target directory in order to make the selection.

- If no files exist within the directory you will need to choose its parent and specify the directory manually by editing the path that appears in the neighbouring text-field. You may even find it quicker to type the whole path name into this field and ignore the file dialog option.

The next action is to specify which operating system the agents will be launched from. Although the agents are completely implemented in Java, and thus will run on any platform with a Java virtual machine, there are subtle differences between Windows and Unix platforms when it comes to creating the command scripts that start the agents.

The command that launches each agent is shown in the "Command Line" field of each agent entry in Figure 6.1. These are the commands that will be written into the scripts during the generation process. From Figure 6.1 it should be evident that the commands to start agents are fairly simple and easy to edit anyway, so the production of scripts is really just provided for the convenience of users.

- To change the type of scripts created, click on either the *Windows* or *Unix* radio buttons.

You are now ready to generate the application source code by clicking on the `Generate` button. Messages reporting on the code generation process will then appear in the "Generation Messages" panel at the bottom of the screen.

Alternatively, pressing the `Cancel` button will dismiss the Code Generation window and nothing will be generated.

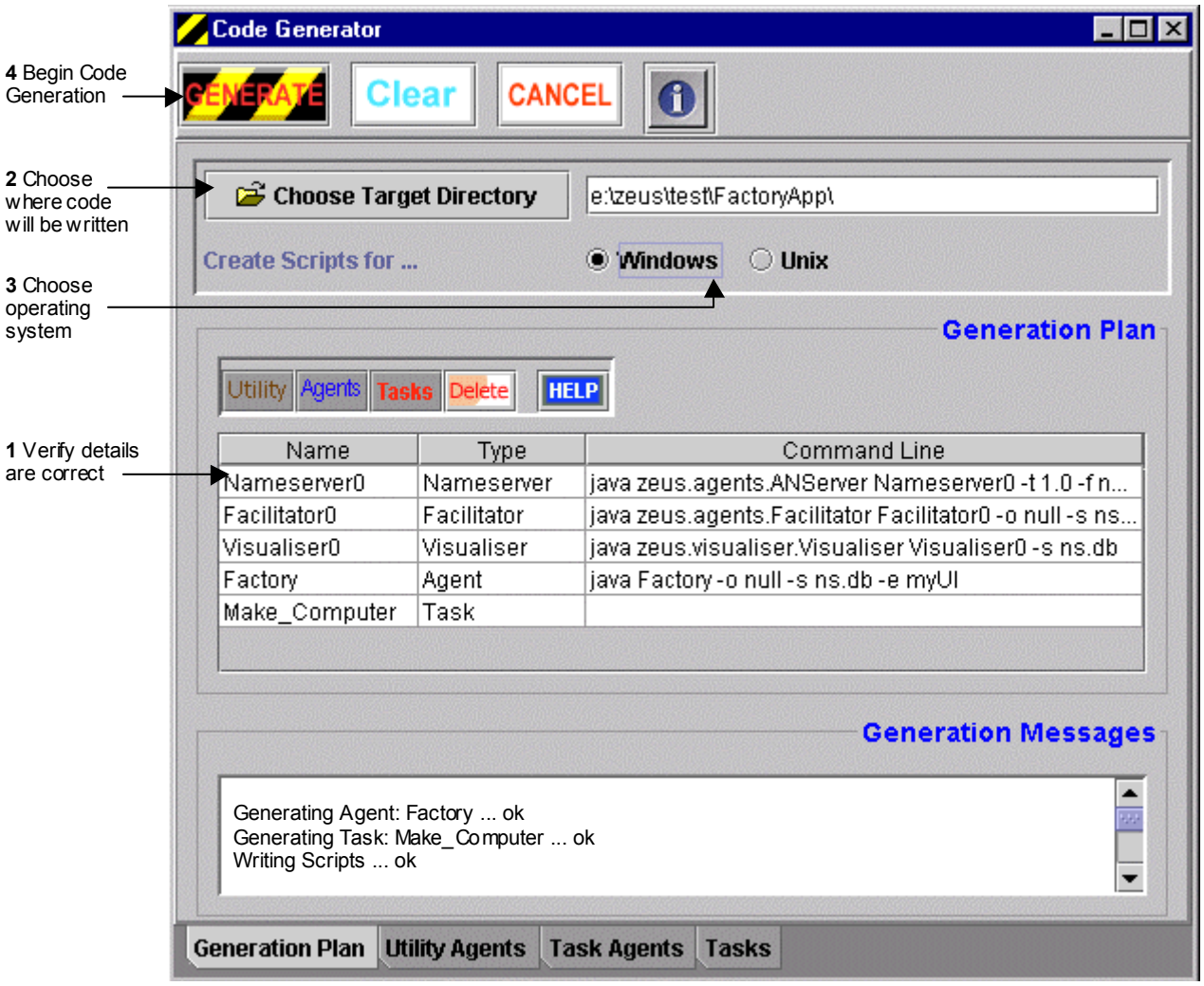


Figure 6.1: The Generation Plan panel, and how to use it

If the generation process is completed successfully the following files will be present in the target directory:

- a Java file for each agent, named <*agentname*>*.java*; these should not be edited
- a Java stub file for each task, named <*taskname*>*.java*; these may need to be edited if any external activities occur when the task is performed, (see activity **IMPL-2**)

Not created are the files that implementing the external resources (see activity **IMPL-3**), and external programs (see activity **IMPL-4**). Instead it is left to developers to implement the application-specific behaviour of these components. The activities that concern the implementation of the stub files and the external components are described during the following sections.

IMPL-2: How to Implement a Task

Tasks are generated in the form of 'stub' files, skeleton implementations that can be augmented by the developer. This enables new domain-specific functionality to be integrated with the automated created agent-specific code, without needing to modify the later.

Figure 6.2 illustrates the role of tasks in relation to agents. In the course of interacting with other agents an agent may need to perform a task, whereupon information is passed from agent to task. Once in the task it can be used for whatever purpose is required, before the information is returned to the agent (modified or not).

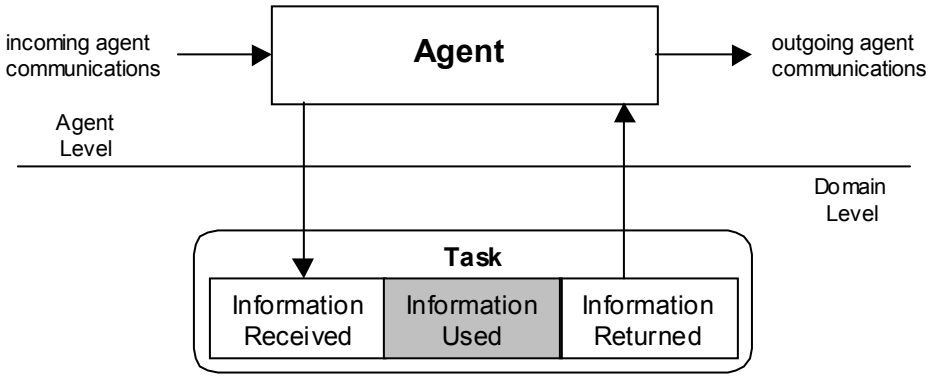


Figure 6.2: The flow of information between an agent and a task

As Figure 6.2 suggests, each task consists of three sections: an initial section where the information is read in from the agent, a mid-section where it is used and a final section when it is returned to the agent. As agents are unaware of what happens inside tasks, when one is performed the agent will pass information into it regardless of whether it will be used or not. Furthermore the agent will expect to receive information in return. The interface between agent and task, i.e. the facts sent and expected back is what is specified when the developer describes the preconditions and effects of a task, (see activity **TASK-1**).

Consequently, of the three sections in a task the mid-section is optional and the first and last sections are generated by default in the task stub file. This means that if the task does not need to use the information passed into it, the developer need not alter the task stub file. Thus this activity is only relevant if the developer wishes to alter how the information received is used, and its scope is limited to the shaded area depicted in Figure 6.2.

A typical task implementation is shown in Figure 6.3; here the points to note are:

- the interface between agent and task is the *exec()* method
- information is passed to the task in the form of arrays of facts. This enables multiple fact instances to be passed into the task when there is precondition fact whose cardinality is greater than 1.
- the information the task expects to receive, i.e. that was specified during the task definition process (**TASK-1**), is passed via the array *expInputArgs*. The information that is *actually* received is passed via the *inputArgs* array; the former can be used to validate the latter.
- the facts that will be returned to the agent are created inside the task using the corresponding expected output arguments in the *expOutputArgs* array as a template.
- the user code occurs after the parameters have been read from the agent and the expected output has been created. In this case the task results are not being modified, only passed to a separate GUI frame for display.
- the created facts are returned to the agent by copying them into the *outputArgs* array. Hence if the task is to modify the information it should be done before this point, otherwise the changes will not be passed to the agent.

SOURCE CODE

```
import java.util.*;
import zeus.util.*;
import zeus.concepts.*;
import zeus.actors.ZeusTask;

public class MakeComputer extends ZeusTask
{
   protected void exec()
   {
      // The Input Facts:
      Fact[] _kb = inputArgs[0];      // KeyBoard
      Fact[] _printer = inputArgs[1]; // Printer
      Fact[] _monitor = inputArgs[2]; // Monitor
      Fact[] _cpu = inputArgs[3];     // CPU

      // The Output Facts:
      Fact[] _computer;       // Computer

      // READ INFORMATION FROM AGENT ------------------------------

      System.out.println("-Expected Input-");
      for(int i = 0; i < expInputArgs.length; i++ )
         System.out.println(expInputArgs[i].pprint());

      System.out.println("-Input-");
      for(int j = 0; j < inputArgs.length; j++) {
         System.  out.println("Input Fact["+j+"]");
         for(int i = 0; i < inputArgs[j].length; i++)
            System.out.println(inputArgs[j][i].pprint());
      }

      System.out.println("-Expected Output-");
      for(int i = 0; i < expOutputArgs.length; i++ )
         System.out.println(expOutputArgs[i].pprint());

      System.out.println("-Output-");
      _computer = new Fact[1];
      _computer[0] = new Fact(Fact.FACT,expOutputArgs[0]);
      System.out.println(_computer[0].pprint());

      /* USER CODE STARTS */

      new DisplayFrame(_computer);

      /* USER CODE ENDS */

      // RETURN INFORMATION TO AGENT ------------------------------

      outputArgs = new Fact[1][];
      outputArgs[0] = _computer;
   }
}
```

Figure 6.3: The Java source code of a typical task body

Typical actions that are encoded into task bodies are displaying information for the user's benefit, writing activity to a log file, or passing information to an external class for processing. The latter case should not be confused with the external programs, (see activity **IMPL-4**), which are called when the agent starts, rather than when tasks are executed.

Once implemented the task should be compiled normally; the resulting .class file must be present in the same directory as the owning agent's .class file for the agent to operate successfully.

IMPL-3: How to Connect an Agent to an External Resource

If an agent needs a resource it will first examine its local resource database, and if the necessary fact is not present it will consult its external resource database (if it has one). These external resources are not generated by the Code Generator, but written by the developer to encapsulate some non-agent storage system, like a database or external program. These are connected to individual agents by implementing the methods of the *zeus.actors.ExternalDb* interface, as shown in Figure 6.4.

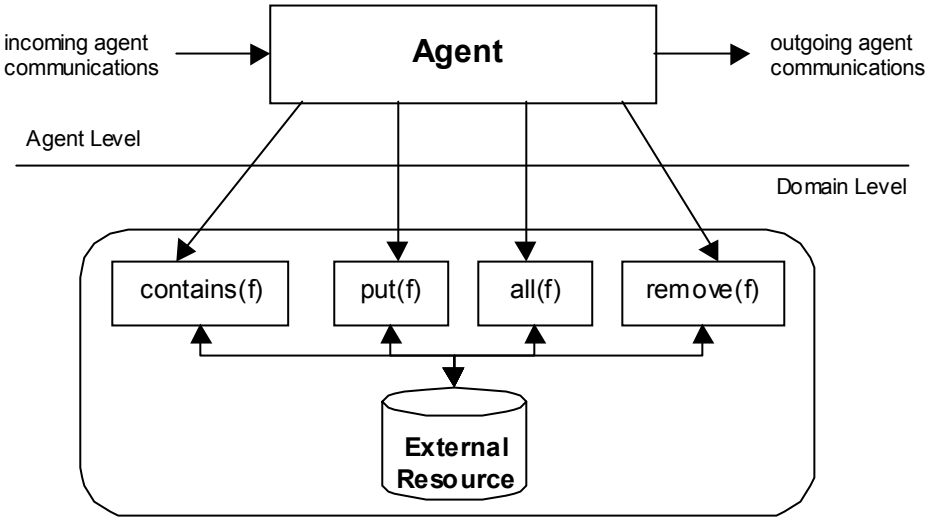


Figure 6.4: The role of ExternalDb is as an interface between an agent and an external resource

To implement a new external resource interface you will need to create a new file in the same directory as the code for the agent that calls it. The file should have the same name that was entered into the agent's `External Database` field, (see activity **TACF-1**).

Looking at the `ExternalDb` interface you will see five abstract methods that need to be implemented. The first is the configuration method, called `set`, which is used to associate the external resource with its owner agent. The others are 'accessor' methods that enable the agent to access the contents of the external resource, these are:

Method	Operation Type	Returns
Contains(Fact)	Membership	Boolean: true if the fact parameter currently exists within the external resource
put(Fact)	Insertion	Boolean: true if the fact parameter was successfully inserted into the external resource. Whether duplicates are permitted is at the discretion of the external resource that implements this service.
all(Fact)	Querying	An enumeration of all facts matching the parameter; this is not assumed to be destructive
Remove(Fact)	Retrieval	The fact that matches the one supplied as the parameter; this is assumed to be destructive

Consequently this activity involves writing the implementations of these methods. The methods illustrated in Figure 6.5 are part of an example application that connects a SQL employee database to an agent. The JDBC methods that enable access to the database return information in the form of strings, as a result the methods must create new fact objects using the attributes retrieved. Also notice how the SQL commands such as "select" and "delete from" are used to access the functionality of the database.

SOURCE CODE

```
package zeus.test.myapp;

import java.util.*;
import java.sql.*;
import zeus.util.*;
import zeus.concepts.*;
import zeus.actors.*;


public userDBwrapper implements ExternalDb
{
  protected AgentContext context;
  protected userDB employeeDB = new userDB();
  protected ResultSet rs;

  public void set(AgentContext ac)
  {
    // agent context enables access to internal agent components
    context = ac;
    employeeDB.openDB();
  }

  public boolean contains(Fact f1)
  {
    String id = f1.getId();
    String query = "select * from employees where empid = '"+id+"'";
    rs = employeeDB.executeQuery(query);
    if (rs.next() == null) return false;
    return true;
  }

   public boolean put(Fact f1)
   {
     String id = f1.getId();
     String position = f1.getValue("position");
     String values = '"+ id + "','" + position +"'";
     employeeDB.executeUpdate("INSERT INTO employees VALUES(" +values+ "')");
     return contains(f1);
   }

   public Fact remove(Fact f1)
   {
     String id = f1.getId();
     String query = "select * from employees where empid = '"+id+"'";
     rs = employeeDB.executeQuery(query);
     if (rs.next() == null) return null;

     Fact newfact = ZeusParser.fact(context.OntologyDb(),
                    "(:type employee :id " + rs.getString(1) +
                    ":attributes ((unit_cost 0)(number 1)(position" +
                    rs.getString(2) + ")))");

     employeeDB.executeUpdate("delete from employees where empid = '"+id+"'");
     return newfact;
   }

   public Enumeration all(Fact f1)
   {
     String id = f1.getId();
     String query = "select * from employees where empid = '"+id+"'";
     rs = employeeDB.executeQuery(query);
     .
     // parse retrieved entries and convert into fact objects
     .
     return facts;
   }
}
```

the object that uses JDBC to encapsulate the database

obtains key attribute from fact to index the database

extracts attributes from fact and inserts them into new entry

information extracted from database and used to build a fact

similiar to remove() except all matches retrieved and nothing is deleted

Figure 6.5: The Java source code of a typical external resource interface

Once implemented the external database interface should be compiled normally and the resulting .class file placed in the same directory as the owning agent's .class file.

IMPL-4: How to Connect an Agent to an External Program

External programs, like external resources, are linked to agents through an interface: in this case *zeus.agents.ZeusExternal*. Thus in this context the term 'external program' is taken to mean any application specific code or non-agent software system that has been encapsulated by an implementation of this interface.

External Programs differ from external resources and task implementations in the conditions that cause them to become active and their duration. External resources are typically invoked when a fact of a particular type was sought, and only active whilst the required information is being retrieved from the resource. Likewise tasks are invoked when a particular activity is performed and terminate when the activity is completed.

By contrast, external programs are launched when the agent is started and may persist for as long as the agent does. This arises from the fact that the external program interface was originally created to enable user interfaces to be connected to agents. However external programs do not necessarily have to be user interfaces, since because the interface provides access to the agent's internal components it would be possible to link systems that can monitor or influence agent behaviour.

The key to linking an external piece of software to an agent is the process by which information in the external code is passed to some internal agent component, where it will have its effect. This requires some knowledge of the interfaces of the agent components. Whilst these interfaces are not hidden (you can see the agent components' methods by looking in the source code of the *zeus.actors* package), using the interfaces correctly does require some insight into the process. This section describes some of the most typical interactions between agent and external program, these are:

- ⇒ How to instruct an agent to do something, (**IMPL-4A**)
- ⇒ How to influence an agent's behaviour, (**IMPL-4B**)
- ⇒ How to access an agent's resources, (**IMPL-4C**)

IMPL-4A: How to Instruct an Agent to Do Something

This form of interaction is obviously involved when the agent receives instructions from a user interface option. (In this case the external program is analogous to the call-back methods commonly used in GUI programming to act upon GUI events).

The actions of ZEUS agents are driven by their internal goals, (an approach known within AI as goal-driven behaviour); hence to issue an instruction to a ZEUS agent all the developer needs to do is give the agent a new goal to achieve. This process is described below, and illustrated by the source code in Figure 6.6. This example is taken from a manufacturing simulation and implements a GUI with a single button, when this is pressed a new goal will be created and sent to the agent instructing it to make a new 'Computer'.

Encoding Objectives

A goal represents an intention to acquire some resource; hence to create a goal you will need to consider what resource represents the objective in question. Given that the application ontology has already been defined this should not be difficult. Once the desired goal type has been decided, you can edit the source code that implements the external program interface.

1) Add a line that retrieves a description of the fact from the ontology database. Note how the *OntologyDb* is accessed with the VARIABLE flag so a description of the fact is retrieved rather than a specific instance.

Once a reference to the required fact has been created you may wish to specify particular values for each of its attributes: this will restrict the specific instance obtained at runtime to satisfy the goal. (If the goal fact is not refined the agent will secure any fact of the correct target type).

2) To refine the fact, use the `setValue()` method for each the significant attributes.

The fact describes what needs to be achieved, but not when and for whom. Thus in order to create a new goal we must specify some additional information. This information may have been retrieved from a user interface, or may be encoded into the external program.

SOURCE CODE

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import zeus.util.*;
import zeus.concepts.*;
import zeus.agents.*;
import zeus.actors.*;

public class ManufacturePump extends JFrame implements ZeusExternal
{
   protected AgentContext context = null;

   public ManufacturePump() {
      JButton startBtn = new JButton("Next");
      JPanel contentPane = (JPanel) getContentPane();
      contentPane.setLayout(new BorderLayout());
      contentPane.add(startBtn,BorderLayout.CENTER);
      startBtn.addActionListener(new SymAction());
      setVisible(false);
      this.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            System.exit(0);
         }
      });
   }

   public void exec(AgentContext context) {
      this.context = context;
      setVisible(true);
      pack();
   }

   protected class SymAction implements ActionListener
   {
      public void actionPerformed(ActionEvent e)
      {
         if ( context == null ) return;

(1)      Fact fact = context.OntologyDb().getFact(Fact.VARIABLE,"Computer");

(2)      fact.setValue("cpu_speed","266");
         fact.setValue("kb_type","UK");
         fact.setValue("printer_type","laser");
         fact.setValue("monitor_type","vga");
         fact.setValue("no","1");

         int now = (int)context.now();
(3)      Goal g = new Goal(context.newId("goal"), fact, now+8, 0, 1,
            context.whoami(), now+4);
(4)      context.Engine().add(g);
      }
   }
}
```

Figure 6.6: The Java source code of a typical external program, highlighting how the 4 key stages of creating and sending a new goal to an agent.

3) The constructor for goal objects has several parameters, (although not all may be significant depending on the objective being achieved). The first parameter is a call to create a new unique identifier for the goal, whilst the second passes the objective defined in the first two stages.

The values of the other goal parameters will depend on the objective being achieved, and application considerations such as cost and urgency, they are:

- The next parameter is the *End Time*: an integer representing the latest time-grain by which the desired effect should be achieved. This is most conveniently expressed relative to the current time-grain. Obviously, the shorter this value the more urgent the goal.
- The next parameter is *Cost*, this is an integer value that effectively sets the budget for achieving the goal. (You may remember from the task definition activity that tasks can have a

price that their owner charges for their invocation - the interpretation of cost being decided by the developer). If the cost is not significant, 0 can be passed in this parameter.

- The next parameter is *Priority*, this also takes an integer value but is currently not utilised during reasoning, use the default value of 1 in the meantime.
- The next parameter is a reference to the name of the agent that owns the goal; the name of the agent connected to the external interface can be obtained through the `whoami()` method.
- The final parameter refers to the *Confirm Time*: this is an integer representing the latest time-grain by which the requesting agent must inform the performing agent whether it has been awarded the contract. Obviously this must be before the end time, and is most conveniently expressed relative to the current time. If this is not important, a value of -1 can be passed.

Once the goal has been created it can be passed to the agent's co-ordination engine, whereupon the agent will begin to try to satisfy the goal.

4) Using the reference to the co-ordination engine contained in the Agent Context, add a line that will pass it the newly created goal.

Once implemented the external program interface should be compiled normally. The external program's .class file should then be in same directory as the owning agent's .class file.

See also…	**For information on…**
Technical Manual, Section 4	How Goals are Achieved by the Co-ordination Engine

IMPL-4B: How to Influence an Agent's Behaviour

This means of affecting agent behaviour differs subtly from the goal-driven approach, and is only applicable to agents who possess a knowledge base of rules. The rule base is a collection of pattern-action rules, when a fact is found that matches an existing pattern, the associated action is performed. Hence this approach involves supplying the agent with facts that will deliberately trigger a response in the agent, this is known as data-driven behaviour.

This will be documented in a future release.

This section will provide a description of how to add actions such as advertising.

IMPL-4C: How to Access an Agent's Resources

One good reason for wanting to access an agent's resource is to inspect its values. For instance, an external program such as a user interface may be used to read and display the attributes of particular resources. Alternatively, you may want modify a resource by changing some of its attributes. This is a much quicker means of changing an agent's internal state than the usual means of sending a message to the agent. Hence this is activity is particularly relevant when a non-agent piece of software needs to affect some change in an agent.

Agent resources (also known as facts) are accessible to the developer by accessing the agent's `ResourceDb` component. The key methods are:

`Fact[] all(String t)`	Returns an array containing all the facts of type **t** currently in the ResourceDb
`void modify(Fact f1, Fact f2)`	Replaces the existing fact **f1** with the fact **f2**; this is typically called after **f1** has been retrieved from the ResourceDb and changed in some respect
`ResourceItem add(Fact f)`	Adds a new fact **f** to the ResourceDb, returning …
`Void del(Fact f)`	Permanently deletes fact **f** from the ResourceDb; this is typically called after a reference to **f** has been obtained using one of the retrieval methods
`Void addFactMonitor(FactMonitor mon, Long eventType)`	Registers a class that implements the FactMonitor interface with the ResourceDb; this fires add, delete and modify events when its state changes.

An excerpt of source code that illustrates how the ResourceDb can be accessed and modified is shown in Figure 6.7. This code is taken from the implementation of a panel that has a label that displays the amount of money currently possessed by the agent. The panel also has a withdrawal button, which when pressed will reset the amount of money held by the agent. The labels in Figure 6.7 are explained below:

1) The moneyPanel class is created, note how it is passed a reference to the AgentContext object of the agent who's resources it will display. The AgentContext object will have been passed by a class that implements the ZeusExternal interface, (see activity **IMPL-4A**).

2) The moneyPanel class registers its interest in ADD fact events that occur within the ResourceDb, (this requires that it implement the methods of the FactMonitor interface). ADD events will be triggered when new facts are added to the ResourceDb or existing facts are modified.

3) The Money fact is read from the ResourceDb, and its value used to initialise the label that displays the amount of money owned by the agent. These lines assume that the agent only has one resource of type 'Money', hence the first element of the array that is returned is used.

4) The actionPerformed method is called when the withdrawal button is pressed, whereupon the reference to the amount of money held by the agent is obtained from the ResourceDb.

5) Setting the Money fact's number attribute to 0 and using it to replace the existing Money resource effectively resets the amount of money held by the agent. It will also cause a fact change event to be triggered.

6) The factAddedEvent method is triggered by the change that occurs in the ResourceDb as a result of (5). In response the money resource is re-read from the ResourceDb and the display label is updated with the new value.

SOURCE CODE

```
package zeus.test.myapp;

import ...

public class moneyPanel extends JPanel implements FactMonitor, ActionListener
{
  protected JLabel  moneyLabel = new JLabel();
  protected JButton resetBtn;

  public moneyPanel(AgentContext agent)                                    (1)
  {
    agent.ResourceDb().addFactMonitor(this, FactEvent.ADD_MASK, true);     (2)
    // .
    // .
    // .
    // code to create panel with label and reset button

    Fact[] tmp = UI.agent.ResourceDb().all("Money");                       (3)
    int num = tmp[0].getNumber();
    moneyLabel.setText("Current balance: " + num);
  }

  public void actionPerformed(ActionEvent evt)
  {
    Fact[] tmp = UI.agent.ResourceDb().all("Money");                       (4)
    if ( tmp.length == 0 ) return;
    Fact f2 = new Fact(tmp[0]);
    f2.setNumber(0);                                                       (5)
    UI.agent.ResourceDb().modify(tmp[0], f2);
  }

  public void factAddedEvent(FactEvent event)                              (6)
  {
    Fact fact = event.getFact();
    int num = fact.getNumber();
    moneyLabel.setText("Current balance: " + num);
  }

  public void factModifiedEvent(FactEvent event) {}
  public void factDeletedEvent(FactEvent event) {}
  public void factAccessedEvent(FactEvent event) {}
}
```

Figure 6.7: An excerpt of source code illustrating how to use and change an agent's resources

IMPL-5: How to Implement a new Interaction Strategy

This will be documented in a future release.

In the meantime you may want to consider the sample negotiation strategies supplied with the toolkit: SimpleInitiatorEvaluator and SimpleRespondentEvaluator, which are implemented in the eponymous files found in the zeus.actors.graphs package. These provide a template from which custom strategies can be created.

Running the Application

Once the agents and any task bodies, external resources and external programs have been implemented and compiled, the next stage is to distribute the agents to their host machines and launch them. This process and the facilities of the Visualisation tools that can be used to inspect and interact with them are described in the final document: the ZEUS Runtime Guide.

Concluding Remarks

This document has described the activities necessary to convert an agent design into a functioning agent. It is part agent cookbook and part ZEUS Generator user manual, providing both a reference of useful expertise that developers can consult to find answers to application implementation problems, whilst also advising how the problem can be solved using the ZEUS Agent Generator. As these solutions become familiar experienced developers will probably consult less document less frequently.

The current release still has a few gaps where toolkit functionality is yet to be documented, but these should be completed in the near future.

In the meantime any errors, comments or suggestions are welcomed.

Jaron Collis (jaron@info.bt.co.uk)